Greedy Heuristics for Judicious Hypergraph Partitioning

Noah Wahl
Karlsruhe Institute of Technology, Germany

Lars Gottesbüren
Karlsruhe Institute of Technology, Germany

Abstract

We investigate the efficacy of greedy heuristics for the judicious hypergraph partitioning problem. In contrast to balanced partitioning problems, the goal of judicious hypergraph partitioning is to minimize the maximum load over all blocks of the partition. We devise strategies for initial partitioning and FM-style post-processing. In combination with a multilevel scheme, they beat the previous state-of-the-art solver – based on greedy set covers – in both running time (two to four orders of magnitude) and solution quality (18% to 45%). A major challenge that makes local greedy approaches difficult to use for this problem is the high frequency of *zero-gain moves*, for which we present and evaluate counteracting mechanisms.

2012 ACM Subject Classification Mathematics of computing → Hypergraphs; Mathematics of computing → Graph algorithms

Keywords and phrases hypergraph partitioning, local search algorithms, load balancing, local search

Digital Object Identifier 10.4230/LIPIcs.SEA.2023.17

Supplementary Material
Software (Source Code): `https://github.com/kahypar/mt-kahypar/tree/judicious_refinement`
Other (Experimental Results and Phylo Instances): `https://github.com/noahares/PhyloBenchmarkSet`

1 Introduction

In this paper, we propose and study greedy heuristics for a variant of hypergraph partitioning named *judicious partitioning* [31], which has applications in load-balanced data distribution, for example in phylogenetic inference [2, 29]. Given a hypergraph $H = (V, E)$ and a number of blocks k, the goal is to partition the nodes V into k disjoint non-empty blocks $V_1, \dots, V_k$, such that the maximum *load* across blocks is minimized. The load $L(V_i)$ of a block V_i is defined as the weight-sum of hyperedges intersecting V_i, i.e., $L(V_i) = \sum_{e \in E, |e \cap V_i| > 0} \omega(e)$.

Contrary to the well-studied balanced partitioning problems with cut-based metrics [10, 23, 27, 14, 17, 15], the judicious variant does not impose a balance constraint on the blocks. Instead, balance is integrated as part of the objective, in the gap between the minimum and maximum load. Yet, just as the balanced variants the judicious partitioning problem is NP-hard [28], such that we focus on heuristics.

Phylogenetic Background

Phylogenetic inference takes a multiple sequence alignment (MSA) as input and tries to derive a *phylogenetic tree*, which is a strictly binary, unrooted tree that estimates the shared evolutionary history of the input. A potential tree topology is scored via a phylogenetic likelihood function (PLF) to estimate the likelihood of the tree, given the MSA. An MSA is a set of n strings from the DNA alphabet (A, T, C, G) and gap-characters such that all strings have the same length l and some distance function is minimized between pairs of strings.

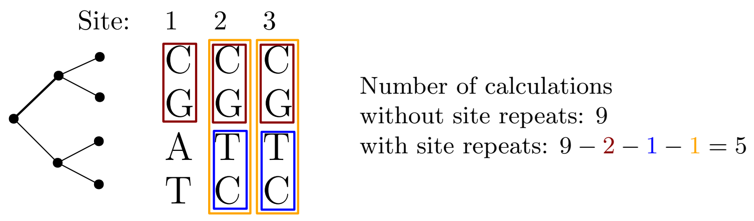

Figure 1 Number of calculations for the PLF with and without site repeats.

It can be thought of as an $n \times l$ matrix where the strings form the rows. A single column of the MSA is called a *site*. To allow for different model parameters (e.g. different genes that evolve at different rates), the sites are split into p disjoint partitions. Because only the strings at the leaves are fixed, the likelihoods of all possible assignments of individual sites at inner nodes have to be calculated under the parameter model(s) to find the likelihood of the whole tree. This is computationally infeasible, so conditional likelihoods for each of the 4 possible characters at each individual site are calculated in a post-order traversal of the tree and combined at the root. However, this still incurs a lot of computation and accounts for 85-95% of total running time of phylogenetic inference tools. In Figure 1 the 4 MSA strings represent the leaves of the proposed tree on the left. For example, at the parent of CCC and GGG, the conditional likelihood for the first site asks for the likelihood of being assigned to A, C, T or G respectively, given that the children are fixed to C and G. For site 1 at the root of this example, the conditional likelihood given that the characters of the first site of the leaves are C, G, A and T, has to be calculated. This results in a total of $3 * 3 = 9$ conditional likelihood calculations for this tree. To parallelize the calculation of the PLF, sites can be split across cores because per-site likelihood calculations are independent of each other. Hence, we need to compute an assignment of sites to cores that minimizes load imbalance. So far, splitting a partition between cores incurs redundant calculations for the model parameters, but sites have equal costs.

The site repeats technique [24] is an optimization to eliminate redundant calculations. It identifies repeating patterns (repeat classes) in parts of distinct sites such that intermediate results can be reused among multiple sites, if they share the same partition and are assigned to the same core (otherwise the results would need to be communicated between cores which adds a scheduling component to the problem). This leads to varying costs for each site in a partition and makes it significantly more difficult to establish load balance between cores. Figure 1 shows an example of site repeats in a single partition that is assigned to a single core. Reusing the results for the pairs C-G (dark red), T-C (blue) and the quadruple C-G-T-C (orange) reduces the number of calculations to 5. Therefore, the goal is to assign sites to cores such that the maximum load is as small as possible, and to keep redundant calculations low, due to repeats split across different cores. Modeled as a hypergraph, each site is a hypernode and each repeats class is a hyperedge. As each hyperedge counts once towards the block-load, this corresponds to judicious partitioning.

Bottleneck Objectives

Objective functions where the value is obtained by taking the maximum across blocks are called *bottleneck* objectives; another example is maximum communication volume, where the maximum cut of edges from a block is minimized. These objectives are particularly challenging for greedy local search heuristics, because all node moves that do not involve the maximum load block do not change the objective function at all. Research on this problem has so far been focused on extremal results [5], particularly for special classes such as bounded degree [7] or uniform hypergraphs [6, 19, 22]. We are only aware of one algorithm, that by Tan et al. [31], and one publicly available implementation thereof called HyperPhylo [2], which improves upon Tan et al.'s work via parallelization and several instance-specific optimizations. Roughly speaking, the idea is to enumerate increasing objective values and determine via a reduction to set cover whether a solution with this load exists. The resulting set cover is then transformed to a node-partition with this load. The set cover problem is solved greedily, which makes suboptimal solutions possible.

While there is only a small amount of literature on judicious partitioning, there is a vast amount on cut-based partitioning. We refer to recent surveys [8, 27, 9] for a broad overview. In this field, the most successful approaches are based on greedy heuristics, which motivates our study in this paper.

Contributions

Despite the difficulties faced by greedy heuristics, we demonstrate that when combined, our approaches significantly beat the existing state-of-the-art algorithm [2] both in terms of objective value (between 18% - 45%) and execution time (between two and four orders of magnitude). Our technical contributions are an iterative improvement algorithm inspired by the classical FM local search [13] (described in Section 3), as well as three greedy construction heuristics (Section 4). We show that randomized repetitions are a simple but effective technique to improve the solution quality and deal with the issue of many *zero-gain moves* to choose from during initial partitioning. Additionally, we considered a simple tie-breaking scheme which favors more balanced loads, but show that it does not lead to improved solutions. To address the issue of scalability of direct k-way initial partitioning for large k, we show that recursive partitioning is a viable option for many types of hypergraphs, but struggles with the class of regular hypergraphs that are encountered in data distribution problems for phylogenetic inference. Furthermore, we integrate our approaches in a state-of-the-art multilevel solver for balanced partitioning [16], leveraging its existing coarsening algorithms to obtain a multilevel solver for judicious partitioning.

Outline

For each component, we conduct thorough experiments on configuration and design choices, before comparing the full system with HyperPhylo in Section 5. In Section 2 we introduce preliminaries, including experimental setup. Each algorithmic component description is directly followed by evaluation and configuration experiments for said component, due to the large number of parameters. Only the best performing configuration moves on to the next section. We refrain from discussing the bio-informatics application in detail, and instead refer to the HyperPhylo paper [2] which describes the connection in detail. In the same vein, we do not conduct parallel phylogenetic inference simulations. Rather, we compare with HyperPhylo in terms of objective values on their benchmark set.

2 Preliminaries

A weighted hypergraph $H = (V, E, w)$ is defined as a set of nodes V and a set of hyperedges $E \subseteq 2^{|V|}$ with hyperedge weights $w : E \to \mathbb{R}_{>0}$. Let $n := |V|$ and $m := |E|$ denote the number of nodes and hyperedges. Functions on sets of nodes or hyperedges are extended to the sum over the set, e.g., $w(T) := \sum_{e \in T} w(e)$ for $T \subseteq E$. The nodes of a hyperedge are called its pins. A node v is called incident to a hyperedge e if $v \in e$. $I(v)$ is the set of all incident hyperedges of v and $|I(v)|$ is the degree of v. By $p = \sum_{e \in E} |e| = \sum_{v \in V} |I(v)|$, we denote the number of pins. Furthermore, we use $[r]$ to denote $[r] \coloneqq \{1, \dots r\}$ for $r \in \mathbb{N}$.

Partitions

A k-way partition is a surjective function $\Pi : V \to [k]$. The blocks $V_i \coloneqq \Pi^{-1}(i)$ of Π are the inverse images. A 2-way partition is also called a bipartition.

The number of pins of a hyperedge e in block V_i is denoted by $\Phi(e, V_i) \coloneqq |V_i \cap e|$. For a block V_i, its *load* is defined as $L(V_i) \coloneqq w(\{e \in E \mid \Phi(e, V_i) > 0\})$. In the *judicious partitioning problem*, the goal is to minimize $max(L(V_1), \dots, L(V_k))$, the maximum load across blocks, which we also call the *judicious load*.

Node Moves: Penalties and Benefits

Our algorithms are based on node moves, i.e., reassigning a given node u from its current block $\Pi(u) = s$ to a different block $\Pi(u) \leftarrow t$. To calculate the difference in the objective function, we use two terms: the benefit and penalties. The *benefit* of removing a node from its current block is $b(u) = w(\{e \in I(u) \mid \Phi(e, \Pi(u)) = 1\})$, i.e., the weight of hyperedges e for which u is the last pin in the block. The *penalty* for adding node u to a given target block t is $p_t(u) = w(\{e \in I(u) \mid \Phi(e, t) = 0\})$, i.e., the weight of hyperedges e which did not intersect the block before, but will now. These values can be efficiently updated after a move, see [26, 18] for more details, where they are used for cut-based objectives. The running time for moving all nodes once with gain updates is $O(pk)$.

Performance Profile Plots

To compare the solution quality of different algorithms, we use *performance profiles* [12]. Let $\mathcal{A}$ be the set of algorithms we want to compare, $\mathcal{I}$ the set of instances, and $q_A(I)$ the maximum load of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. For each algorithm A, we plot the fraction of instances (y-axis) for which $q_A(I) \leq \tau \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)$, where τ is on the x-axis. Achieving higher fractions at lower τ-values is considered better. For $\tau = 1$, the y-value indicates the percentage of instances for which an algorithm performs best.

Machine Setup

All experiments are run on an AMD EPYC Rome 7702P with 2x64 cores clocked at 2.0-3.35 GHz with 1024 GB DDR4 RAM at 3200 MHz. Our proposed algorithms are single-threaded. The only parallelism used in our solver is during coarsening and for randomized repetitions during initial partitioning. Coarsening usually has negligible running time, and in the main experiments we use at most 5 repetitions. For HyperPhylo we used all 128 cores.

Benchmark Sets

We use two separate established benchmark sets for our experimental evaluations, which we refer to as set A and set P. The input instances are unweighted, however during multilevel coarsening hyperedges are aggregated and thus receive non-uniform weights.

Set A [21] consists of 488 real-world hypergraphs from different application domains for cut-based hypergraph partitioning, such as VLSI design [32, 1], SAT solving [4], and sparse matrices [11]. It is available from `https://algo2.iti.kit.edu/schlag/sea2017/` in hMetis format [23]. The hypergraphs in set A contain between 6K - 100M pins, 160 - 13M hyperedges and 7K - 13M nodes, with more detailed statistics available on the website.

Set P is derived from the data of Baar et.al. [2] used in their evaluation of HyperPhylo. It consists of a total of 11 hypergraphs; 7 smaller hypergraphs derived from sequence data from collaborative studies with biologists (prefixed with 59, 128 and 404 in our experiments) [30] and 4 larger hypergraphs from the one thousand insect transcriptome evolution project (the so called *supermatrix*, prefixed with sm in out experiments) [25]. An important property of the hypergraphs of set P is that they are regular, i.e. all nodes have the same degree, because each site has exactly one repeats class per inner node of the phylogenetic tree. These instances are available from their repository at `https://github.com/lukashuebner/HyperPhylo`. We converted these graphs to the hMetis format and made them available at `https://github.com/noahares/PhyloBenchmarkSet`.

We predominantly use set A for configuration experiments, due to its size and variety of hypergraphs. Set P is used to verify our results for these regular graphs, so we can later use our best configuration for the comparison with HyperPhylo. We include results for set P in our experimental sections alongside results on set A to show significant differences. The horse-race comparison with HyperPhylo in Section 5 is conducted only on set P, since HyperPhylo requires uniform node degrees (this is an implementation restriction to enable some optimizations).

3 Iterative Improvement

In this section, we introduce our first algorithmic contribution, namely an iterative improvement algorithm. To refine an initial partition we employ a local moving strategy similar to FM [13]. The full algorithm is shown in Algorithm 1. Contrary to non-bottleneck objectives, the only way to improve judicious load directly is to move nodes out of the block with the highest load. Let us denote this block by V_s. The order in which nodes are moved is prioritized by a gain function g defined in Equation 1, which represents the difference in load if a node $u \in V_s$ is moved to a different block V_t. In each step, we determine the highest gain node u and associated target block t, see line 4 in Algorithm 1. We then move u from s to t and update their loads.

$$g(u,t) = \begin{cases} b(u) & \text{if } L(V_t) + p_t(u) \leq L(V_s) - b(u) \\ L(V_s) - L(V_t) - p_t(u) & \text{else} \end{cases} \tag{1}$$

There are three possible scenarios for the maximum load after a move. If V_s remains the heaviest block, the load is decreased by $b(u)$. Otherwise, if V_t becomes the new heaviest block, then $b(u)$ has no influence because we only care about by how much $L(V_t) + p_t(u)$ differs from the prior maximum load $L(V_s)$. Furthermore, let V_i be the block with the highest load other than V_s. If $L(V_s) - b(u) < L(V_i)$, the actual gain is additionally capped at $L(V_i) - L(V_s) + b(u)$, because V_i becomes the new highest load block. To encourage some further optimization on V_s before moving on to V_i, we ignore this cap when calculating the next move. However, if the target block V_t becomes the new heaviest block through the move, the gain is negative, and we finish the optimization on V_s, see line 5.

In line 9, we update the gain values $b(v), p_s(v), p_t(v)$ for neighbors v with $\{u, v\} \subset e$ such that $\Phi(e, s) \leq 1$ or $\Phi(e, t) = 1$ (after the move). These are (almost) the same updates as for the cut-based objectives, see for example [16, 27] for details (we can omit an update for $\Phi(e, t) = 2$). In the worst case, the total cost of updates across the entire run is $O(kp)$, but in practice we often observe behavior closer to $O(p)$ since the average number of blocks intersecting a hyperedge is close to constant. To determine the highest gain move, we use one priority queue per target block, which we now update to reflect the new gain values of neighbors after the move.

Algorithm 1 Judicious FM.

```
1  while $\max_{i\in[k]}(L(V_i)) > \min_{i\in[k]}(L(V_i)) \cdot \delta$ do
2      $s \leftarrow \arg\max_{i\in[k]}(L(V_i))$
3      while $\max_{i\in[k]-s}(L(V_i)) < L(V_s) \cdot \gamma$ do
4          $(u,t) \leftarrow \arg\max_{v\in V_s, i\in[k]-s}(g(v,i))$
5          if $g(u,t) < 0$ then break
6          $\Pi(u) \leftarrow t$
7          $L(V_s) \leftarrow L(V_s) - b(u)$
8          $L(V_t) \leftarrow L(V_t) + p_t(u)$
9          run gain updates for neighbors of $u$
10     if $s = \arg\max_{i\in[k]}(L(V_i))$ then break
```

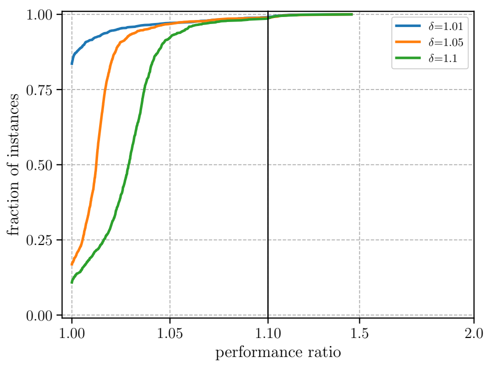

Figure 2 Performance profiles for varying δ in judicious FM on set A.

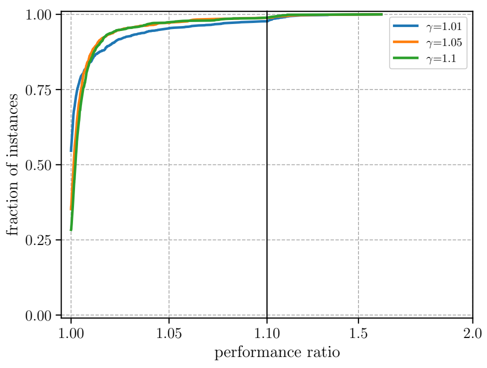

Figure 3 Performance profiles for varying γ in judicious FM on set A.

We continue the inner loop at line 3 until V_s is no longer the block with the highest load, subject to some relative margin $\gamma > 1$. This encourages further optimization on V_s and avoids frequently alternating between the two highest loaded blocks. However, once V_s is no longer the highest load block, we can worsen the solution (due to the ignored additional cap). This happens for example when all blocks have similar loads. As mentioned, we thus also break out of the loop once only negative gain moves remain.

We continue the outer loop (line 1) as long as there is a block with smaller load that can take in new nodes, again subject to a relative margin parameter $\delta > 1$. Notice that judicious FM cannot achieve loads smaller than the initial minimum load. We experimented with ideas to break out of local minima, but were unsuccessful. Thus, an important consideration for initial partitioning heuristics is to produce solutions with a considerable gap between minimum and maximum load.

Furthermore, notice that δ offers a trade-off between solution quality and running time, where smaller values give better loads, but need to perform more optimization. We note however, that, since we quickly steer into a local minimum, the running time of FM is usually negligible compared to initial partitioning.

3.1 Experiments

In the following, we evaluate the impact of the two parameters γ and δ, as well as how much judicious FM improves over the initial partition. For δ we expect small values to give the best solutions, whereas for γ the picture is not as clear. We want the other heaviest block to

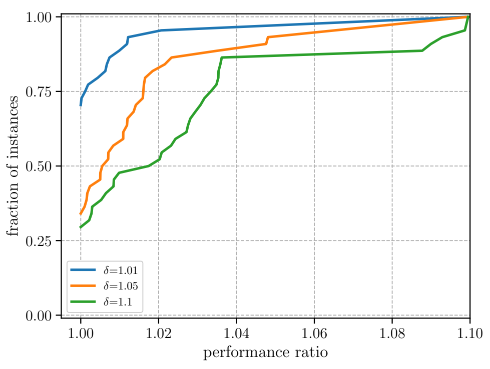

Figure 4 Performance profiles for varying δ in judicious FM on set P.

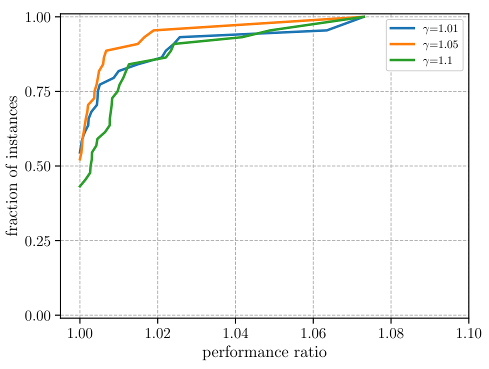

Figure 5 Performance profiles for varying γ in judicious FM on set P.

Table 1 Mean and percentile of the load achieved by judicious FM divided by the load of the initial solutions.

mean	gmean	p_{10}	p_{25}	p_{50}	p_{75}	p_{90}	max
1.43323	1.34457	1.05399	1.10473	1.21581	1.45799	1.89561	11.45786

be significantly heavier than the current one, to enable more improvement before we switch blocks again. However, if the gap between minimum and maximum load is small, there is not much room for improvement, and we are quickly left with only negative gains.

For both parameters, we tried the values $1.01, 1.05$ and 1.1. We ran a quadratic grid-search, but for readability the following figures display one parameter varied at a time, with the other fixed to the best value. Figure 2 (set A) and Figure 4 (set P) demonstrate that smaller values indeed perform better for δ. As the refinement scheme converges very quickly, its running time is often negligible compared to initial partitioning, so we use $\delta = 1.01$ in the following. In Figure 5, we see that $\gamma = 1.05$ performs best for set P, whereas Figure 3 shows that there is no significant difference for set A. We observed that the inner loop is more frequently terminated by negative gains than the stopping condition, which explains this behavior.

Finally, in Table 1 we show average and percentile improvements over the initial solution, that is the ratio between the load achieved by judicious FM divided by that of the initial solution. We see a geometric mean improvement of 34.4% and a median of 21.5% from using FM. These values are similar to what we can expect in cut-based partitioning [3]. The initial solutions for these experiments were obtained with the best performing configuration for initial partitioning from Section 4, which we discuss next.

4 Initial Partitioning

In this section, we present our greedy initial partitioning algorithms and evaluate their performance. We start with an approach where we construct all k blocks at the same time, and subsequently leverage the presented strategies as a subroutine for recursive partitioning.

4.1 Direct k-way Initial Partitioning

In initial partitioning, we start with all nodes unassigned and do not move already assigned nodes (until the refinement stage later on).

Similar to FM, the order in which nodes are assigned is determined greedily via a loss function $\varrho(v,i)$. At each step, we determine the node and target block with the lowest loss, and then update the unassigned nodes' losses. The difference is that all nodes must be moved (assigned) at the end. In the following we propose three strategies to define ϱ. Since nodes are initially unassigned, the loss functions are not based on benefit values $b(v)$ as in the move gain, just penalty terms $p_t(v)$ are incorporated. The time complexity of all three variants is $O(\log(n)(n+p)k)$. This bound stems from the updates to penalty values after an assignment [26, 18], and the priority queue updates.

Penalty

In the penalty strategy, we use $\varrho(v,t) = p_t(v)$, i.e., the next node to be assigned has the lowest $p_t(v)$ value globally. In this strategy, nodes are attracted to blocks that already contain many of its neighbors. Thus, already highly loaded blocks are preferred. As we do not impose a constraint on the block size or load, there is no mechanism to achieve a balanced distribution of nodes, as we would need for the traditional balanced partitioning problem. Despite this, it is an interesting strategy to consider, as it nudges the solution into very different local minima than the following strategies, which focus more on balanced blocks. While not competitive on the initial solutions, on some instances we observed better solutions after FM refinement, because the gap between the minimum and maximum load is wider, and thus the refinement has more room for improvement.

Block Load

In the block load strategy, we first select the currently lightest block t, and only then choose the node v with minimal $p_t(v)$. The advantage over the penalty strategy is that it keeps block loads evenly distributed. This comes at the cost of not considering good moves to blocks that are not the lightest.

Judicious Increase

In the judicious increase strategy, the loss of assigning a node v to block t is $\varrho(v,t) = \max(L(V_t) + p_t(v) - \max_{i\in[k]}(L(V_i)),\ 0)$. It keeps balance in mind by trying to increase the loads of all blocks evenly. Since this loss definition directly corresponds to the loss in maximum load, we expected the judicious increase strategy to perform the best, which is confirmed in the experiments. Yet, we decided to also evaluate the other two strategies as the judicious increase strategy is particularly prone to long streaks of consecutive equal losses, which makes it hard to distinguish between the different possible assignments to perform next.

Furthermore, we investigate a simple tie-breaking scheme for the judicious increase strategy. Allowing losses to become negative by omitting the outer *max* introduces tie-breaking of zero-loss moves by $L(V_t) + p_t(v)$. Intuitively speaking, this optimizes for more balanced loads, if we are not adding to the heaviest block.

4.2 Randomization

We use priority queues for selecting the target block first and then the node to move to that target block. Because all three strategies result in many equal loss-scores, the order in which moves are chosen heavily depends on the order of insertions into the priority queues. To break this dependency we introduce randomization, combined with multiple differently seeded repetitions, to cover a variety of possible assignment sequences. We implement this by attaching a randomly generated tag to each move added to a priority queue, which is used as the secondary comparison criterion. The same approach can be used for choosing different blocks in case of ties, to prevent assigning all nodes to one single block. Randomization and repetitions result in significantly increased odds of finding a good initial partition.

4.3 Coarsening

One of the most important components in cut-based partitioning is the multilevel scheme [20]. Initial partitioning and iterative improvement are not run directly on the input hypergraph. The hypergraph is iteratively coarsened by repeatedly contracting node clusters, which approximately preserve the structure, until the hypergraph is fairly small. Each iteration and associated hypergraph constitutes a level in the hierarchy. On the lowest level (the smallest hypergraph) an initial partition is computed, which is then projected back through the hierarchy, by assigning clustered nodes to the same block. Furthermore, iterative improvement is run on each level.

While designed for cut-based objectives, this scheme is directly applicable to the judicious partitioning problem, with a small modification. Hyperedges of size 1 cannot be removed, since they still contribute to the volume of their pin's block. To speed up the algorithmic components, we still remove such hyperedges and track the removed volume at each node.

We stop coarsening (to transition into initial partitioning) once the current hypergraph has less than $C \cdot k$ nodes, for a constant parameter C. This ensures, that we can place C nodes in each block on average, and thus the optimization algorithms have some leeway. In the following experiments we use $C = 50$, which was determined in preliminary experiments that are omitted here. The multilevel scheme has two major advantages over flat partitioning: iterative improvement can optimize the solution at different levels of granularity, and it is substantially faster since initial partitioning runs on small inputs.

4.4 Experiments on Direct k-way Initial Partitioning

For the evaluation of our direct k-way initial partitioning strategies we expected the judicious increase strategy to clearly outperform the other two. Figure 6 indeed shows a wide gap between the strategies. However, we were also interested in the trade-off between load-balanced initial partitions versus creating more room for improvement in the refinement phase. Figure 7 initially shows potential for the penalty strategy combined with FM post-processing, as it achieves more of the best solutions than the block-load strategy. However, the curve remains flat, being overtaken by block-load at just the 1.01 mark, indicating that on the solutions where the penalty strategy performs worse, it is significantly worse. On set P, see Figures 8, 9, the results are even more clearly in favor of judicious increase. We conclude that the judicious increase strategy produces the best k-way initial partitioning results, and thus use it as the default algorithm from now. Note that there are only small differences in running time between the strategies, which is why we omit plots for these here.

Next, we look at the effect of randomization on solution quality. This is shown in Figure 10. We clearly see an improvement when using 5 repetitions, but diminishing returns for 10 repetitions. Hence, we use 5 repetitions as the default value from now.

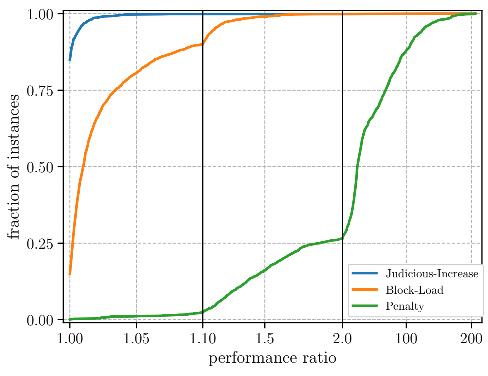

Figure 6 Performance profiles for initial partitioning strategies on set A.

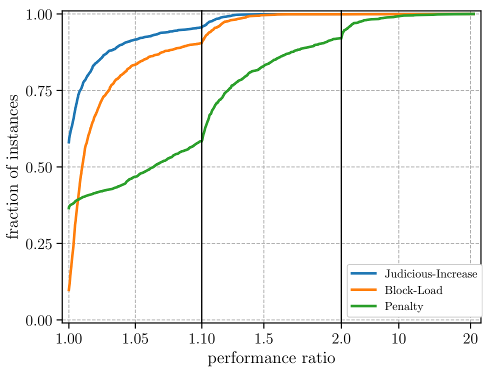

Figure 7 Performance profiles for initial partitioning strategies with FM post-processing on set A.

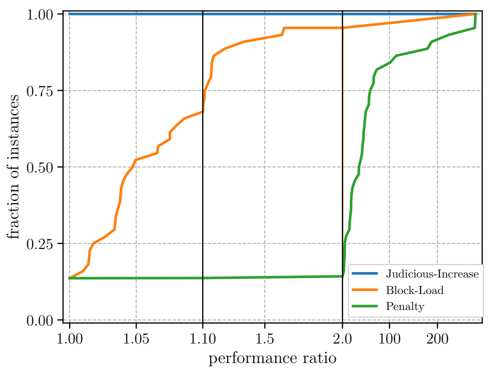

Figure 8 Performance profiles for initial partitioning strategies on set P.

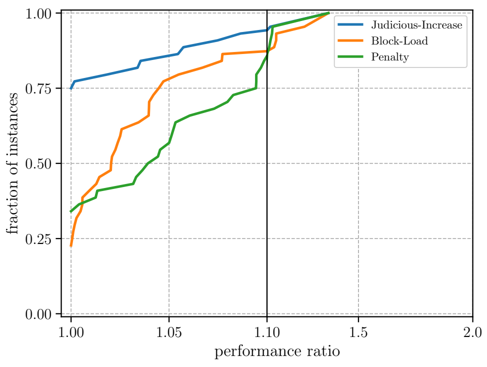

Figure 9 Performance profiles for initial partitioning strategies with FM post-processing on set P.

Lastly, Figure 11 shows that tie-breaking has no effect, neither positive nor negative. We assume that this stems from the fact that more balanced solutions early on inhibit the optimization potential of FM post-processing later on.

Still, we argue that there is a need for tie-breaking schemes. In Table 2 we report in how many steps of the initial partitioning procedure the best loss value is zero for the supermatrix instances of set P. There is a clear correlation between the number of zero loss moves and k. This is expected, as there are more blocks that offer free assignments, before becoming the new heaviest block. While for $k = 48$ only 10-20% of assignments have zero loss, for $k = 2048$ it is 97%-99%. These results indicate that there is certainly a need to distinguish zero-loss moves, but further investigation is needed to find better tie-breaking mechanisms.

4.5 Recursive Partitioning

The difficulty of many ties in the scores when constructing k-way partitions directly, was already observed for cut-based objectives. There, the solution is to restrict the assignment options by recursively bipartitioning on the coarsest graph, and transition to k-way local search only in the uncoarsening phase. This yields better solution quality than direct k-way [27] for initial partitions but we will show this is not the case for judicious partitioning. One reason to still consider recursive partitioning is the improved time complexity of $O(p \log k)$ compared to direct k-way's complexity of $O(pk)$.

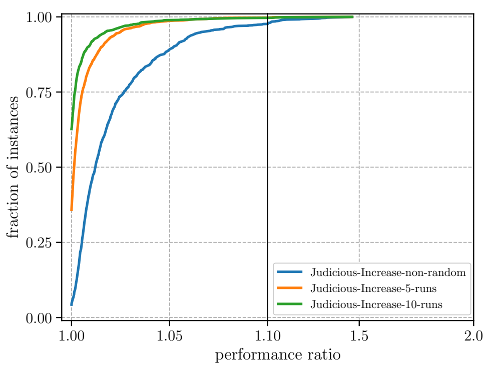

Figure 10 Effect of randomized repetitions on set A.

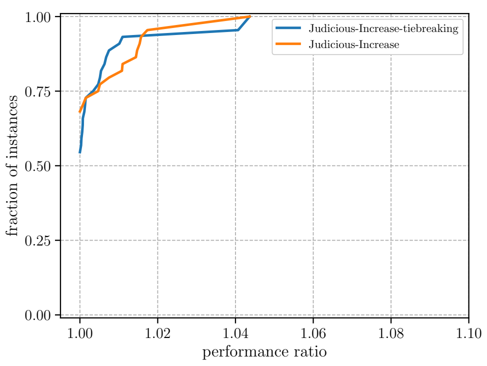

Figure 11 Effect of tie-breaking on set P.

Table 2 Fraction of zero-loss assignments on supermatrix instances of set P for direct k-way initial partitioning with the judicious increase strategy.

k	graph	$\frac{\lvert\varrho(v,t)=0\rvert}{\#moves}$	k	graph	$\frac{\lvert\varrho(v,t)=0\rvert}{\#moves}$
48	sm_part1_170859	0.113	256	sm_part1_170859	0.762
	sm_part3_31854	0.151		sm_part3_31854	0.802
	sm_part12_20753	0.152		sm_part12_20753	0.825
	sm_part24_11756	0.212		sm_part24_11756	0.866
160	sm_part1_170859	0.574	2048	sm_part1_170859	0.991
	sm_part3_31854	0.602		sm_part3_31854	0.984
	sm_part12_20753	0.636		sm_part12_20753	0.979
	sm_part24_11756	0.725		sm_part24_11756	0.976

In recursive bipartitioning, we first split into two blocks, then extract the sub-hypergraphs induced by the two blocks, and recursively partition these, leading to a binary recursion tree of depth $\lceil \log_2(k) \rceil$ to obtain a k-way partition. There are several issues with recursive partitioning, such as not optimizing the objective function directly, and the fact that splitting a node pair is irreversible. On the judicious metric we note one additional problem, namely if k is not a power of 2. Let $k_1 = \lfloor k/2 \rfloor, k_2 = \lceil k/2 \rceil$. We would need to target an imbalance of (k_1/k) to (k_2/k) in the loads of the two blocks, in order to (let us say hand-wavingly) pass load evenly down the recursion. Unfortunately, our algorithms are not designed for this. Instead, we optimize for balanced load on bipartitions (using judicious FM at each level), and assign the smaller number of final blocks k_1 to the recursive call on the block with smaller load. As additional base cases to the recursion, we perform direct 3-way or 5-way partitioning, which are the two cases with the highest required imbalance.

4.6 Experiments on Recursive Partitioning

In Figure 13, we show that direct k-way partitioning (DK) gives better solution quality than recursive bipartitioning (RB) for initial partitioning on set P. On set A the situation is less clear, see Figure 12, where recursive partitioning achieves more of the best solutions, but direct k-way converges faster towards 1. It is unclear why we see such a large discrepancy in initial judicious loads between the benchmark sets. Looking at geometric mean running times on set A, we have $0.039s$ for direct k-way versus $0.29s$ for recursive partitioning with

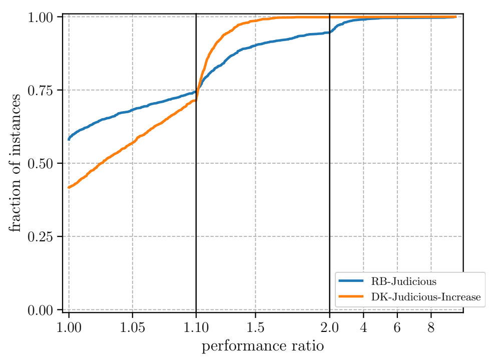

Figure 12 Performance profiles for initial partitioning judicious loads on set A.

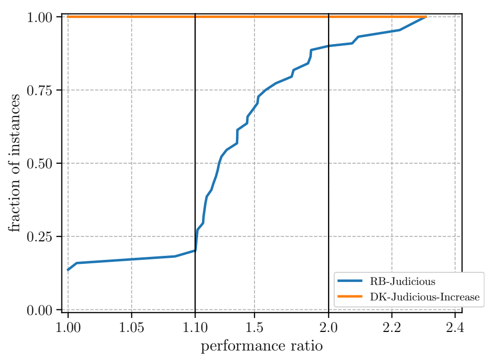

Figure 13 Performance profiles for initial partitioning judicious loads on set P.

$k = 48$, $0.354s$ versus $0.622s$ with $k = 160$, and $0.96s$ versus $0.81s$ with $k = 256$. Recursive partitioning becomes faster at $k = 256$, and for larger k there will be a point where flat direct k-way is no longer feasible. On set P, the geometric mean time for all k is 0.1s for recursive partitioning and 0.05s for direct k-way, and respectively 0.013s versus 0.017s for $k = 2048$. We conclude that direct k-way partitioning is the method of choice over recursive partitioning for initial judicious partitioning, as long as k is moderate.

5 Comparison to HyperPhylo

In this section, we evaluate the performance of our solver against that of HyperPhylo [2], the existing state of the art for judicious partitioning. Our solver uses the multilevel scheme with coarsening down to $50 \cdot k$ nodes, flat k-way partitioning with the judicious increase strategy on the coarsest hypergraph (randomization enabled, tie-breaking disabled), and refinement with judicious FM ($\delta = 1.01, \gamma = 1.05$) on each level.

Since the HyperPhylo implementation[1] is restricted to instances with uniform node degree (for optimization purposes), the comparison is restricted to set P. Table 3 shows loads and running times for our solver, as well as loads and times relative to ours for HyperPhylo (ours divided by HyperPhylo). Values below one thus correspond to cases where our solver performed better. These are highlighted in bold. Cases with $n \leq k$ are omitted because the optimal partition is to place each node in its own block, which is trivial.

The table is split into two parts: smaller instances at the top, and larger instances (supermatrix) at the bottom, which are deemed most important. We see that our solver outperforms HyperPhylo by a substantial margin in terms of running time and load in *all* reported cases on the supermatrix instances. The running time improvements range from two to four orders of magnitude, whereas the load improvements are between 18% to 45%.

On the seven smaller instances, our running time advantage is less pronounced. HyperPhylo even beats our solver on 6 out of 21 runs, particularly for the largest value of $k = 2048$. This is because larger values of k incur smaller maximum loads, which is an advantage of HyperPhylo, since it enumerates increasing objective values. On the other hand, our initial partitioning algorithms become slower, the larger k is, since there are more assignments to evaluate in each step. The largest slowdown is a factor of 59 (50ms vs 3s) for the instance 404-1 with $n = 2161$ on $k = 2048$, where both solvers achieve maximum load 402. This value

[1] `https://github.com/lukashuebner/HyperPhylo`

is the degree lower bound (each node must be assigned to one block), such that HyperPhylo finishes in the first iteration, whereas our solver has to go through all optimization steps. In terms of solution quality, our solver remains superior: it is beaten on only 4 out of 21 runs.

Table 3 Running times and loads of our solver versus HyperPhylo on benchmark set P. Values where our solver performs better are highlighted in bold.

graph	n	m	k	our load	$\frac{\text{our load}}{\text{HyperPhylo}}$	our time [s]	$\frac{\text{our time}}{\text{HyperPhylo}}$
59-s	160	671	48	77	1.35088	0.00450	**0.15511**
128-s	204	1170	48	167	1.06369	0.00648	**0.34105**
			160	126	1.00000	0.01151	1.27943
404-s	588	2525	48	511	**0.99031**	0.05110	**0.68127**
			160	438	1.08955	0.10469	7.47786
			256	402	1.00000	0.12523	8.34880
128-0	857	10853	48	550	**0.59524**	0.02196	**0.01923**
			160	289	**0.54735**	0.04136	**0.04452**
			256	243	**0.71261**	0.05950	**0.12501**
404-l	2161	40648	48	1718	**0.67162**	0.14527	**0.02432**
			160	879	**0.75451**	0.32195	**0.07700**
			256	771	**0.84262**	0.46104	**0.13795**
			2048	402	1.00000	3.09564	59.53160
59-l	2183	10205	48	359	**0.67608**	0.02833	**0.03818**
			160	164	**0.70386**	0.05991	**0.14332**
			256	129	**0.73295**	0.09367	**0.29363**
			2048	57	**0.98276**	1.23261	51.35861
128-l	2933	23618	48	1023	**0.69782**	0.07136	**0.01432**
			160	466	**0.71472**	0.17621	**0.04306**
			256	370	**0.68773**	0.25684	**0.06748**
			2048	204	1.22156	2.18591	2.41271
sm_part24_11756	11756	99713	48	5814	**0.71355**	0.44393	**0.00137**
			160	2436	**0.61702**	0.90976	**0.00307**
			256	1714	**0.61324**	1.83163	**0.00635**
			2048	431	**0.55256**	9.69627	**0.04347**
sm_part12_20753	20753	163514	48	8018	**0.70968**	0.76193	**0.00120**
			160	3331	**0.63255**	1.28946	**0.00225**
			256	2365	**0.59707**	2.20258	**0.00393**
			2048	567	**0.60512**	17.96227	**0.04103**
sm_part3_31854	31854	185662	48	9107	**0.71126**	1.03926	**0.00121**
			160	3902	**0.66474**	1.91767	**0.00247**
			256	2691	**0.65110**	2.64924	**0.00352**
			2048	646	**0.62056**	27.02443	**0.04377**
sm_part1_170859	170859	196836	48	11515	**0.82174**	3.33007	**0.00065**
			160	5104	**0.81209**	5.27129	**0.00107**
			256	3778	**0.76493**	7.45439	**0.00152**
			2048	956	**0.74339**	98.21790	**0.02070**

6 Conclusion and Future Work

In this paper, we designed and evaluated a set of greedy heuristics for the judicious hypergraph partitioning problem, namely an iterative improvement algorithm based on FM, and three initial partitioning strategies. We argued that greedy heuristics face severe challenges (such as equal gains/losses and scalability issues), and presented remedies such as randomization, tie-breaking and recursive partitioning. While these did not work as well as intended, we demonstrate nonetheless that combined with the multilevel framework, our algorithms are faster (two to four orders of magnitude) and yield substantially better solution quality (18% to 45%) than the previous state-of-the-art algorithm.

Future work should focus on ways to escape local minima during refinement (such as simulated annealing), ideas such as higher level gains as tie-breakers for the issue with many equal gains, as well as parallelization. Furthermore, we are interested in evaluating the impact of our approach on applications, such as the phylogenetic inference application that motivated HyperPhylo [2].

References

1 Charles J. Alpert. The ISPD98 Circuit Benchmark Suite. In *International Symposium on Physical Design (ISPD)*, pages 80–85, April 1998. `doi:10.1145/274535.274546`.

2 Ivo Baar, Lukas Hübner, Peter Oettig, Adrian Zapletal, Sebastian Schlag, Alexandros Stamatakis, and Benoit Morel. Data Distribution for Phylogenetic Inference with Site Repeats via Judicious Hypergraph Partitioning. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPSW 2019, Rio de Janeiro, Brazil, May 20-24, 2019*, pages 175–184. IEEE, 2019. `doi:10.1109/IPDPSW.2019.00038`.

3 R Battiti, A Bertossi, and R Rizzi. Randomized Greedy Algorithms for the Hypergraph Partitioning Problem. *Randomization Methods in Algorithm Design, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 43:21–35, 1999. `doi:10.1090/dimacs/043/02`.

4 Anton Belov, Daniel Diepold, Marijn Heule, and Matti Järvisalo. The SAT Competition 2014. `http://www.satcompetition.org/2014/`, 2014.

5 Béla Bollobás and Alex D. Scott. Judicious Partitions of Hypergraphs. *Journal of Combinatorial Theory*, 78(1):15–31, 1997. `doi:10.1006/jcta.1996.2744`.

6 Béla Bollobás and Alex D. Scott. Judicious Partitions of 3-uniform Hypergraphs. *European Journal of Combinatorics*, 21(3):289–300, 2000. `doi:10.1006/eujc.1998.0266`.

7 Béla Bollobás and Alex D. Scott. Judicious partitions of bounded-degree graphs. *Journal of Graph Theory*, 46(2):131–143, 2004. `doi:10.1002/jgt.10174`.

8 Aydin Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering - Selected Results and Surveys*, volume 9220 of *Lecture Notes in Computer Science*, pages 117–158. Springer, 2016. `doi:10.1007/978-3-319-49487-6_4`.

9 Ümit V. Çatalyürek, Karen D. Devine, Marcelo Fonseca Faraj, Lars Gottesbüren, Tobias Heuer, Henning Meyerhenke, Peter Sanders, Sebastian Schlag, Christian Schulz, Daniel Seemaier, and Dorothea Wagner. More Recent Advances in (Hyper) Graph Partitioning. *ACM Computing Surveys*, 2022. `doi:10.1145/3571808`.

10 Ümit V. Catalyurek and Cevdet Aykanat. Hypergraph-Partitioning-based Decomposition for Parallel Sparse-Matrix Vector Multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 10(7):673–693, 1999. `doi:10.1109/71.780863`.

11 Timothy A. Davis and Yifan Hu. The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, 38(1):1:1–1:25, November 2011. `doi:10.1145/2049662.2049663`.

12 Elizabeth D. Dolan and Jorge J. Moré. Benchmarking Optimization Software with Performance Profiles. *Mathematical Programming*, 91(2):201–213, 2002. doi:10.1007/s101070100263.

13 Charles M. Fiduccia and Robert M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Conference on Design Automation (DAC)*, pages 175–181, 1982. doi:10.1145/800263.809204.

14 Lars Gottesbüren, Michael Hamann, Sebastian Schlag, and Dorothea Wagner. Advanced Flow-Based Multilevel Hypergraph Partitioning. *18th International Symposium on Experimental Algorithms (SEA)*, 2020. doi:10.4230/LIPIcs.SEA.2020.11.

15 Lars Gottesbüren, Tobias Heuer, and Peter Sanders. Parallel Flow-Based Hypergraph Partitioning. Technical report, Karlsruhe Institute of Technology, 2022.

16 Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Scalable Shared-Memory Hypergraph Partitioning. In *23st Workshop on Algorithm Engineering & Experiments (ALENEX)*. SIAM, January 2021. doi:10.1137/1.9781611976472.2.

17 Lars Gottesbüren, Tobias Heuer, Peter Sanders, and Sebastian Schlag. Shared-Memory n-level Hypergraph Partitioning. In *24th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM, January 2022. doi:10.1137/1.9781611977042.11.

18 Lars Gottesbüren. *Parallel and Flow-Based High-Quality Hypergraph Partitioning*. Dissertation, Karlsruhe Institute of Technology, 2022.

19 John Haslegrave. The Bollobás-Thomason conjecture for 3-uniform hypergraphs. *Combinatorica*, 32(4):451–471, 2012. doi:10.1007/s00493-012-2696-x.

20 Bruce Hendrickson and Robert W. Leland. A Multi-Level Algorithm For Partitioning Graphs. In Sidney Karin, editor, *Proceedings Supercomputing '95, San Diego, CA, USA, December 4-8, 1995*, page 28. ACM, 1995. doi:10.1145/224170.224228.

21 Tobias Heuer and Sebastian Schlag. Improving Coarsening Schemes for Hypergraph Partitioning by Exploiting Community Structure. In *16th International Symposium on Experimental Algorithms (SEA)*, pages 21:1–21:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, June 2017. doi:10.4230/LIPIcs.SEA.2017.21.

22 Jianfeng Hou, Shufei Wu, and Guiying Yan. On judicious partitions of uniform hypergraphs. *Journal of Combinatorial Theory*, 141:16–32, 2016. doi:10.1016/j.jcta.2016.02.004.

23 George Karypis, Rajat Aggarwal, Vipin Kumar, and Shashi Shekhar. Multilevel Hypergraph Partitioning: Applications in VLSI Domain. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(1):69–79, 1999. doi:10.1109/92.748202.

24 Kassian Kobert, Alexandros Stamatakis, and Tomáš Flouri. Efficient detection of repeating sites to accelerate phylogenetic likelihood calculations. *Systematic biology*, 66(2):205–217, 2017.

25 Bernhard Misof, Shanlin Liu, Karen Meusemann, Ralph S Peters, Alexander Donath, Christoph Mayer, Paul B Frandsen, Jessica Ware, Tomáš Flouri, Rolf G Beutel, et al. Phylogenomics resolves the timing and pattern of insect evolution. *Science*, 346(6210):763–767, 2014.

26 Laura A. Sanchis. Multiple-Way Network Partitioning. *IEEE Transactions on Computers*, 38(1):62–81, 1989. doi:10.1109/12.8730.

27 Sebastian Schlag. *High-Quality Hypergraph Partitioning*. Dissertation, Karlsruhe Institute of Technology, 2020. doi:10.5445/IR/1000105953.

28 Farhad Shahrokhi and László A Székely. The complexity of the bottleneck graph bipartition problem. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 15(94):221–226, 1994.

29 Alexandros Stamatakis. Raxml version 8: a tool for phylogenetic analysis and post-analysis of large phylogenies. *Bioinformatics*, 30(9):1312–1313, 2014. doi:10.1093/bioinformatics/btu033.

30 Alexandros Stamatakis and Nikolaos Alachiotis. Time and memory efficient likelihood-based tree searches on phylogenomic alignments with missing data. *Bioinformatics*, 26(12):i132–i139, 2010.

31 Tunzi Tan, Jihong Gui, Sainan Wang, Suixiang Gao, and Wenguo Yang. An Efficient Algorithm for Judicious Partition of Hypergraphs. In *Combinatorial Optimization and Applications - 11th International Conference, COCOA 2017, Shanghai, China, December 16-18, 2017, Proceedings, Part II*, volume 10628 of *Lecture Notes in Computer Science*, pages 466–474. Springer, 2017. doi:10.1007/978-3-319-71147-8_33.

32 Natarajan Viswanathan, Charles J. Alpert, Cliff C. N. Sze, Zhuo Li, and Yaoguang Wei. The DAC 2012 Routability-Driven Placement Contest and Benchmark Suite. In *49th Conference on Design Automation (DAC)*, pages 774–782. ACM, June 2012. doi:10.1145/2228360.2228500.