Contents

1 Notation

Let A be some set. We will denote the set $\{\{a_1, \ldots, a_k\} \subseteq A : a_1, \ldots, a_k \text{ pairwise distinct}\}$, which contains all k-ary subsets of A, by $\binom{A}{k}$. For the same set A, P_A denotes the set of all partitions of A. For a partition $\Pi \in P_A$ and an element $a \in A$, $[a]_\Pi$ denotes the set in Π that contains a, of which there is exactly one. If A is finite, $\mathcal{U}(A)$ denotes the uniform distribution over A, i.e. the distribution that assigns every element in A the same probability (namely: $1/|A|$). If we want to indicate that a random variable $\mathbf{X}$ has probability distribution $\mathcal{Q}$, we will write $\mathbf{X} \sim \mathcal{Q}$. If f is a function from the domain of a random variable $\mathbf{X} \sim \mathcal{Q}$ to the real numbers, its expected value is written as $\mathbb{E}_{\mathbf{X}\sim\mathcal{Q}}[f(\mathbf{X})]$ and defined as $\mathbb{E}_{\mathbf{X}\sim\mathcal{Q}}[f(\mathbf{X})] = \sum_{i=1}^{n} \mathcal{Q}(x_i) f(x_i)$, if the domain of $\mathbf{X}$ is finite and given by $\{x_1, \ldots, x_n\}$ (we will only consider finite domains).

2 Introduction

This work is concerned with the partioning problem with respect to a given finite set V, with a cost structure defined over 3-ary subsets of V. The considered problem is hard in many respects: one, the amount of possible partitions grows large very fast (with growing $|V|$). Two, even computing the objective function for a given partition is prohibitive, since summing over all 3-ary subsets takes almost $|V|^3$ steps (and to add to that, not even $|V|^2$ steps are feasible if $|V|$ becomes larger). Thus, we are interested in a way of approximately solving this problem by the use of local search algorithms and some considerations on the objective function. Since the No-Free-Lunch-Theorem (Wolpert et al., [1]) implies that there is no such thing as the "best" local search algorithm for arbitrary cost-structures, we want to omit too much focus on specific search algorithms and rather focus on some shared difficulties, e.g. the efficient...

- ...representation of partitions,
- ...enumeration of "neighbours" for a given partition (possibly in random order) and
- ...computation (or estimation) of the given cost function and cost-improvements.

The problem is given as follows. Let $V = \{v_1, \ldots, v_n\}$ be a finite set, which we will call the set of vertices. Associated with this set are functions

$$c, c' : \binom{V}{3} \mapsto \mathbb{R}$$

which define an arbitrary cost-structure on 3-ary subsets of V. For a 3-ary subset $\{u, v, w\} \in \binom{V}{3}$ and a given partition $\Pi \in P_V$, we define the cost of $\{u, v, w\}$ with respect to Π as

$$\ell(\{u,v,w\}, \Pi) = \begin{cases} c(\{u,v,w\}) & \text{if } [u]_\Pi \neq [v]_\Pi, [u]_\Pi \neq [w]_\Pi, [w]_\Pi \neq [v]_\Pi \\ c'(\{u,v,w\}) & \text{if } [u]_\Pi = [v]_\Pi = [w]_\Pi \\ 0 & \text{otherwise.} \end{cases}$$

This can be interpreted as follows: whenever u, v and w are part of pairwise different sets in Π, the cost of $\{u, v, w\}$ is equal to the costs as defined by c. If they are part of the same set, the cost of $\{u, v, w\}$ is equal to the costs as defined by c'. Otherwise, if neither of the above is the case, the cost of $\{u, v, w\}$ is just 0. Based on this definition, we are confronted with problems of the form

$$\Pi^* = \arg\min_{\Pi \in P_V} \sum\nolimits_{\{u,v,w\} \in \binom{V}{3}} \ell(\{u,v,w\}, \Pi)$$

i.e. we wish to find a partition Π^* of V that minimizes some objective function over 3-ary subsets of V. In the following section 3, we will discuss an efficient way of representing partitions, e.g. with respect to their space requirements. Afterwards, in section 4, we consider ways of transforming these representation such that we are able to efficiently generate a neighbourhood (in random order) for each partition. In section 5 we will investigate how the change in the objective function can be computed or estimated, with respect to the neighbourhood of a given partition. The penultimate part 6 contains a theoretic discussion on how the proposed settings influence the runtime of a simple hill-climbing algorithm, which will be evaluated in section 7.

3 Indexings as Proxies for Partitions

Since we want to use a local search algorithm that iteratively finds better solutions (partitions in this case), we want to consider on how we want to represent said partitions. One may represent a partition as a set of sets of vertices, which is probably the most straight-forward approach. On the one hand, the required space is linear in $|V|$. But on the other hand, this comes with a few problems: checking if two vertices are part of the same set, and the removal and the addition of a vertex from or to a set all require $|V|$ steps in the worst case (if all vertices have the same set). A last problem is that one has to gurantee feasibility at every step, i.e. that every solution is indeed a partition. A second idea might be to store a partition as a $|V| \times |V|$ boolean matrix, where every row corresponds to a vertex and every column to a set in the partition, and an entry at position i, j would mean "vertex i is part of set j" if 1 and "vertex i is not part of set j" if 0. This would alleviate some of the above problems, since looking up, removal or addition of vertices to sets would take constant time in the worst case. However, storing a partition would require $|V|^2$ units of space, which might become problematic when $|V|$ is large or when considering multiple partitions at once.

Therefore, we will use the following concept of indexings (which induce equivalence relations on V) to solve the above problems, i.e. to simplify and speed up operations on partitions. Every vertex is mapped to a number (index) which indicates the set in the partition we want this vertex to be a part of. If multiple vertices are mapped to the same index, they will be part of the same set in the partition. Since there are at maximum $|V|$ sets in a partition of V (every vertex is assigned its own set, i.e. the indexing is bijective), we restrict to indices between 1 and $|V|$. Note that every indexing can be stored in space $O(|V|)$.

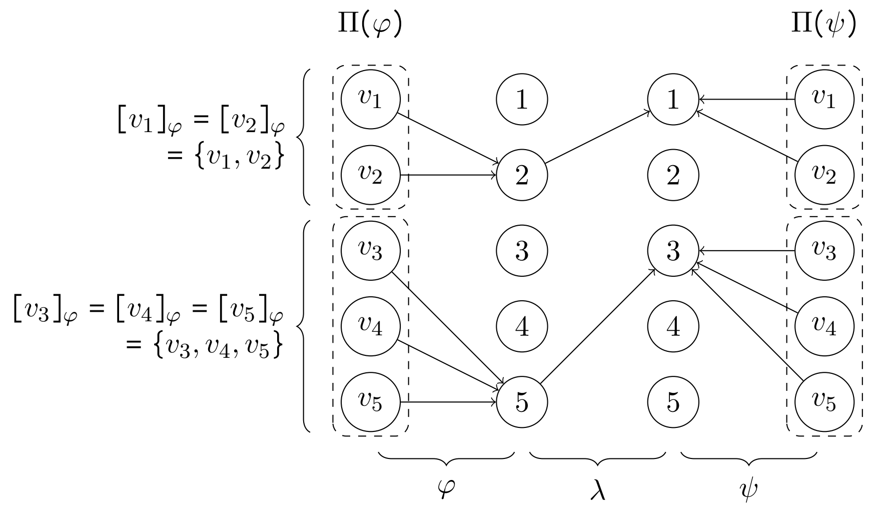
Definition 3.1. *Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of vertices. An indexing of V is a mapping $\varphi \in \{1, \ldots, n\}^V = [n]^V$ that associates every vertex with a number from 1 to n.*

Definition 3.2. *Let φ be an indexing of V. The partition induced by φ is defined as*

$$\Pi(\varphi) = \{[v]_\varphi : v \in V\}, \tag{3.1}$$

where $[v]_\varphi = \{w \in V : \varphi(w) = \varphi(v)\}$.

Based on definition 3.1 and 3.2, we are able to obtain some immediate results. First, we are able to use indexings as representations for partitions, i.e. for every partition, there is some



$\Pi(\varphi)$
$\Pi(\psi)$
$[v_1]_\varphi = [v_2]_\varphi = \{v_1, v_2\}$
$[v_3]_\varphi = [v_4]_\varphi = [v_5]_\varphi = \{v_3, v_4, v_5\}$
v_1 v_2 v_3 v_4 v_5
1 2 3 4 5
φ
λ
ψ

Figure 1: Illustration of two indexings φ, ψ with corresponding partitions and proof of equality through part (b) of Theorem 3.1.

indexing that yields that very partition (Lemma 3.1). And two, we obtain a criteria that makes it possible to check when two vertices share the same set in the partition (Lemma 3.2).

Lemma 3.1. *Π partitions V if and only if there exists an indexing of V that induces Π.*

Proof. First, we will show the direction from left to right. Let $\Pi = \{U_1, \dots, U_m\}$, $1 \leq m \leq n$, be a partition of V. For all vertices v with corresponding set U_i in Π (of which there is exactly one), we define $\varphi(v) = i$. For a $j \in \{1, \dots, m\}$ we then obtain $\varphi^{-1}(j) = U_j$. But then $\Pi(\varphi) = \Pi$, i.e. Π is induced by φ.
The other direction requires us to prove that every indexing induces a partition of V. I.e., for an indexing φ of V with $\Pi(\varphi) = \Pi$, we need to check the following requirements:

1. $\varnothing \notin \Pi$: This is obvious from (3.1), since every element $[v]_\varphi \in \Pi$ contains v.
2. $\bigcup_{U \in \Pi} U = V$: "$\subseteq$" is obvious. For "$\supseteq$", take a vertex v. Then, $v \in [v]_\varphi$ and since $[v]_\varphi \in \Pi(\varphi)$, we obtain $v \in \bigcup_{U \in \Pi} U$.
3. if $U_1, U_2 \in \Pi$ and $U_1 \neq U_2$, then $U_1 \cap U_2 = \varnothing$: Take two $U_1, U_2 \in \Pi$ with $U_1 \neq U_2$ such that $U_1 = [w]_\varphi$ and $U_2 = [u]_\varphi$ for two vertices w and u. Assume for a contradiction that $U_1 \cap U_2 \neq \varnothing$, i.e. there exists $v \in U_1 \cap U_2$. But then $v \in [w]_\varphi$ and $v \in [u]_\varphi$. Thus, $\varphi(v) = \varphi(w) = \varphi(u)$, which cannot be the case, since that would imply $U_1 = U_2$.

This completes the proof. □

Lemma 3.2. *Let φ be an indexing of V. For all vertices v, u, we have $\varphi(v) = \varphi(u)$ if and only if v and u are part of the same set in $\Pi(\varphi)$.*

Proof. For the direction from left to right, take $v, u \in V$ such that $\varphi(v) = \varphi(u)$. Then $v, u \in [v]_\varphi = [u]_\varphi \in \Pi(\varphi)$, i.e. they are part of the same set. For the other direction, we will show the contraposition. Take $v, u \in V$ such that $\varphi(v) \neq \varphi(u)$. Clearly, $v \in [v]_\varphi$ and $v \notin [u]_\varphi$ as well as $u \in [u]_\varphi$ and $u \notin [v]_\varphi$. Therefore $[v]_\varphi \neq [u]_\varphi$. Since by Lemma 3.1,

$\Pi(\varphi)$ is a partition and $[v]_\varphi$ and $[u]_\varphi$ are distinct sets in $\Pi(\varphi)$, $[v]_\varphi$ is the only set that contains v and vice versa for $[u]_\varphi$ and u. Thus, v and u cannot be part of the same set in $\Pi(\varphi)$. □

Since we want to use indexings in order to define transformations on partitions, we are interested in the question when two indexings are "equal", in the sense that they induce the same partition. This is characterized in part by Theorem 3.1.

Theorem 3.1. *If φ, ψ are two indexings of V, then the following statements are equivalent:*

(a) $\Pi(\varphi) = \Pi(\psi)$

(b) there is a bijection $\lambda : \text{image}(\varphi) \to \text{image}(\psi)$[1] *such that* $\lambda(\varphi(v)) = \psi(v)$ *for all* $v \in V$

(c) for all vertices v, w, $\varphi(w) = \varphi(v)$ if and only if $\psi(w) = \psi(v)$

Proof. We will start with the direction from (a) to (b). Let φ, ψ be two indexings of V with $\Pi(\varphi) = \Pi(\psi)$. We define $\lambda : \text{image}(\varphi) \to \text{image}(\psi)$ as follows. For all $k \in \text{image}(\varphi)$, associate a vertex v such that $\varphi(v) = k$. Then define $\lambda(k) = \psi(v)$. It remains to show that λ fulfills the requirements in (b):

1. λ is bijective: Since every element in $\text{image}(\varphi)$ corresponds to a set in $\Pi(\varphi)$, and vice versa for $\text{image}(\psi)$ and $\Pi(\psi)$, we get $|\text{image}(\varphi)| = |\Pi(\varphi)| = |\Pi(\psi)| = |\text{image}(\psi)|$. Since $\text{image}(\varphi)$ and $\text{image}(\psi)$ are also finite, it suffices to show injectivity of λ in order to prove bijectivity. Assume for a contradiction that λ is not injective, i.e. there are $v_1, v_2 \in V, \varphi(v_1) \neq \varphi(v_2)$, such that $\lambda(\varphi(v_1)) = \lambda(\varphi(v_2)) = \psi(v_1) = \psi(v_2)$. Thus, by application of Lemma 3.2, there must be a set in $\Pi(\psi)$ that contains v_1 and v_2, while there is no set in $\Pi(\varphi)$ that has both vertices in it. But then $\Pi(\varphi) \neq \Pi(\psi)$. Contradiction.

2. For all vertices v, $\lambda(\varphi(v)) = \psi(v)$: Take an arbitrary vertex v and let $\varphi(v) = k \in \text{image}(\varphi)$. Let $\bar{v}$ be the vertex that was previously associated with k in the definition of λ, i.e. the vertex for which $\varphi(\bar{v}) = k = \varphi(v)$ and $\lambda(\varphi(\bar{v})) = \psi(\bar{v})$ holds. Assume for a contradiction that $\psi(v) \neq \psi(\bar{v})$. By Lemma 3.2, this yields that there is no set in $\Pi(\psi)$ that contains both v and $\bar{v}$, while there is one in $\Pi(\varphi)$. But then again $\Pi(\varphi) \neq \Pi(\psi)$, i.e. a contradiction. Thus, $\lambda(\varphi(v)) = \lambda(\varphi(\bar{v})) = \psi(\bar{v}) = \psi(v)$.

This concludes this direction. For the direction from (b) to (c), let λ be a bijection between the images of two indexings φ, ψ of V that fulfills the requirements in (b). Let w, v be two arbitrary vertices; then

$$\begin{aligned} \varphi(w) = \varphi(v) \quad &\text{iff} \quad \lambda(\varphi(w)) = \lambda(\varphi(v)) && (*) \\ &\text{iff} \quad \psi(w) = \psi(v) && (**) \end{aligned}$$

The first identity $(*)$ works since λ is a bijection and the second $(**)$ since $\lambda(\varphi(w)) = \psi(w)$

[1] $\text{image}(\varphi)$ means the image of φ, i.e. $\text{image}(\varphi) = \{\varphi(v)\}_{v \in V}$.

for all vertices w. The remainder (c) to (a) is relatively trivial: If we assume premise (c), then

$$\begin{aligned} U \in \Pi(\varphi) \quad &\text{iff} \quad U = [v]_\varphi \text{ for a vertex } v \\ &\text{iff} \quad U = \{w \in V : \varphi(w) = \varphi(v)\} \text{ for a vertex } v \\ &\text{iff} \quad U = \{w \in V : \psi(w) = \psi(v)\} \text{ for a vertex } v \\ &\text{iff} \quad U = [v]_\psi \text{ for a vertex } v \\ &\text{iff} \quad U \in \Pi(\psi), \end{aligned}$$

which shows $\Pi(\varphi) = \Pi(\psi)$. □

If the task is to determine whether two induced partitions $\Pi(\varphi), \Pi(\psi)$ for given indexings φ, ψ of V are equal, the result of Theorem 3.1 might be useful: instead of explicitly computing the resulting partitions and checking if both sets are equal, one can simply determine whether a fitting bijection λ exists, which is arguably easier. In fact, one can construct λ as in the first part of the proof of Lemma 3.1 and then check whether the result is a bijection or not. Figure 1 illustrates two indexings, their partitions and some of the findings of Theorem 3.1.

4 Move-Operation on Indexings

We will now define the "move"-operation on indexings, which transforms an indexing into another indexing by changing the assigned index of a vertex. The effect this has on the induced partition is that the respective vertex is "taken" from its original set[2] and then "put into" some other set. This is then defined as

$$\varphi_{v \to k}(u) = \begin{cases} \varphi(u) & u \neq v, \\ k & u = v \end{cases} \tag{4.1}$$

for every vertex u. The operation takes as input an indexing φ, a vertex v and new index k and outputs a new indexing that is essentially the same as φ, with the only difference being that v is mapped to k instead of whatever it was mapped to before.

4.1 Move-Enumeration

In many cases, there are multiple ways of moving one vertex to different indices while still inducing the same partition afterwards. For example, if v is an arbitrary vertex, φ is an indexing of V with pairwise different k_1, k_2 such that there is no vertex u with $\varphi(u) = k_1$ or $\varphi(u) = k_2$, then moving v to k_1 or k_2 yields that while $\varphi_{v \to k_1}$ and $\varphi_{v \to k_2}$ are different indexings, their induced partitions are the same: $\Pi(\varphi_{v \to k_1}) = \Pi(\varphi_{v \to k_2})$. Based on this observation we are interested in an efficient way of enumerating all possible "moves" of a vertex with respect to the induced partitions without enumerating too much or having to double-check whether two move-operations induce the same partition. Hence, consider

[2] i.e. the set $[v]_\varphi$

algorithm 1, which aims at finding a solution to this problem.

Algorithm 1: Move-Enumeration

```
Input: Set of vertices $V$ with indexing $\varphi$
Result: Sequence of moves $(w_1, k_1), \dots, (w_m, k_m)$
1  Let $<$ be some linear order on $V$
2  Let $\mathcal{N} := \{1, \dots, n\} \backslash \mathrm{image}(\varphi)$
3  forall vertices $w \in V$ do
4      forall $\varphi(v) \in \mathrm{image}(\varphi) \backslash \{\varphi(w)\}$ do
5          if $[w]_\varphi = \{w, u, \dots\}$ or $[v]_\varphi = \{v, s, \dots\}$ then
6              enumerate $(w, \varphi(v))$
7          else if $[w]_\varphi = \{w\}$ and $[v]_\varphi = \{v\}$ and $w < v$ then
8              enumerate $(w, \varphi(v))$
9      if $[w]_\varphi = \{w, u, v, \dots\}$ or $([w]_\varphi = \{w, u\}$ and $w < u)$ then
10         Let $k \in \mathcal{N}$
11         enumerate $(w, k)$
```

The intuition is as follows. Line 1 fixes a linear order on V, which can be understood as a preference relation when the movement of two different vertices would yield the same partition. Line 2 defines $\mathcal{N}$ as the container for all indices that have no assigned vertices. What follows is the consideration of all possible moves of a vertex v to some other index k (line 3, 4, 9): in line 4, vertices are possibly moved to indices that are not empty, and in line 9, vertices are possibly moved to empty indices[3].

We want to check three important properties: one, algorithm 1 only enumerates moves which yield pairwise distinct partitions (the "not too much"-part), two, every possible move is equivalent to one of the enumerated moves and three, the induced partition by any enumerated move is not the same as the partition induced by φ. I.e., if φ is an indexing of V and $(w_1, k_1), \dots, (w_m, k_m)$ is a sequence generated by algorithm 1 on input V and φ, Lemma 4.1, 4.2 and 4.3 hold.

Lemma 4.1 (Pairwise Distinctiveness). *$\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$ for all $1 \leq i < j \leq m$.*

Proof. Note that every pair $(w, k) \in V \times \{1, \dots, n\}$ is regarded at most once. This means that for every two different indexings $\varphi_{w_i \to k_i}, \varphi_{w_j \to k_j}$, either $w_i \neq w_j$ or $k_i \neq k_j$ holds. Now, pick $i, j \in \{1, \dots, m\}$ such that $i \neq j$. The rest of the proof can be done via case distinction on all possible conditions under which a move could be enumerated in the algorithm:

1. $w_i = w_j = w$. Since i and j are pairwise distinct, $k_i \neq k_j$ must hold. For k_i, we have either $k_i \in \mathcal{N}$ (line 9) or $k_i = \varphi(v) \in \mathrm{image}(\varphi) \backslash \{\varphi(w)\}$ (line 6 and 8):

   (a) If $k_i \in \mathcal{N}$: Then $\varphi_{w_i \to k_i}(w) = k_i$ only for w (since there is no vertex that is mapped to k_i in φ). Also $k_j \notin \mathcal{N}$, since there is at maximum one $k \in \mathcal{N}$ for which $\varphi_{w \to k}$ is enumerated. Thus, $k_j = \varphi(v) \in \mathrm{image}(\varphi) \backslash \{\varphi(w)\}$. But then we have $\varphi_{w_j \to k_j}(w) = k_j = \varphi_{w_j \to k_j}(v)$ and $\varphi_{w_i \to k_i}(v) = k_j \neq k_i = \varphi_{w_i \to k_i}(w)$, which implies $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$ by Theorem 3.1.

   (b) If $k_i = \varphi(v) \in \mathrm{image}(\varphi) \backslash \{\varphi(w)\}$: After moving w to k_i in $\varphi_{w_i \to k_i}$, one obtains $\varphi_{w_i \to k_i}(w) = \varphi_{w_i \to k_i}(v) = k_i$, and after moving w to k_j in $\varphi_{w_j \to k_j}$, one gets

[3] Mind that if $\mathcal{N} = \varnothing$, then there is no way line 9 will evaluate to true: since every index has some vertex that is assigned to it, and the amount of vertices is equal to the amount of indices, every vertex is "alone" at its index.

$\varphi_{w_j \to k_j}(w) = k_j \neq k_i = \varphi_{w_j \to k_j}(v)$. But that implies $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$ by Theorem 3.1.

2. $w_i \neq w_j$, and therefore $\varphi_{w_i \to k_i}(w_j) = \varphi(w_j)$ and $\varphi_{w_j \to k_j}(w_i) = \varphi(w_i)$ (i.e., $\varphi_{w_i \to k_i}$ does not change the index of w_j and $\varphi_{w_j \to k_j}$ does not change the index of w_i). Again, we make the distinction for the case $k_i \in \mathcal{N}$ and $k_i = \varphi(v) \in \text{image}(\varphi) \setminus \{\varphi(w)\}$:

   (a) If $k_i \in \mathcal{N}$: since $k_i \notin \text{image}(\varphi)$, $\varphi(w_i) \neq k_i$. Also, at least one of the following cases (see line 9 of the algorithm) must hold:

      i. If $[w_i]_\varphi = \{w_i, u, v, \dots\}$: At least one of the vertices u and v must be different from w_j, since $u \neq v$ and w_j cannot be equal to both of them. Let w.l.o.g. $u \neq w_j$. Then $\varphi_{w_j \to k_j}(w_i) = \varphi(w_i) = \varphi(u) = \varphi_{w_j \to k_j}(u)$ but $\varphi_{w_i \to k_i}(w_i) = k_i \neq \varphi(w_i) = \varphi_{w_i \to k_i}(u)$, i.e. $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$ by Theorem 3.1.

      ii. If $[w_i]_\varphi = \{w_i, u\}$ and $w_i < u$:

         A. $w_j \neq u$: We get $\varphi_{w_i \to k_i}(w_i) \neq \varphi_{w_i \to k_i}(u)$ and $\varphi_{w_j \to k_j}(w_i) = \varphi(w_i) = \varphi(u) = \varphi_{w_j \to k_j}(u)$. Then simply apply Theorem 3.1 and obtain $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$.

         B. $w_j = u$. Since $u \nless w_i$ and $[w_j]_\varphi = [w_i]_\varphi$, there is no possibility that line 11 is executed for w_j. Therefore, $k_j \notin \mathcal{N}$. But then $k_j = \varphi(v) \in \text{image}(\varphi) \setminus \{\varphi(w_j)\}$ with $w_i \neq v \neq w_j$. Thus, $\varphi_{w_j \to k_j}(w_j) = \varphi(v) = \varphi_{w_j \to k_j}(v)$ and $\varphi_{w_i \to k_i}(w_j) = \varphi(w_j) \neq \varphi(v) = \varphi_{w_i \to k_i}(v)$. Again, the application of Theorem 3.1 yields $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$.

   (b) If $k_i = \varphi(v) \in \text{image}(\varphi) \setminus \{\varphi(w_i)\}$, then one of the following cases applies:

      i. $[w_i]_\varphi = \{w_i, u, \dots\}$.

         A. If $w_j = u$: then $\varphi_{w_j \to k_j}(w_i) = \varphi(w_i) \neq \varphi(v) = \varphi_{w_j \to k_j}(v)$ and $\varphi_{w_i \to k_i}(w_i) = \varphi(v) = \varphi_{w_i \to k_i}(v)$. Here, Theorem 3.1 can be applied, which results in $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$.

         B. If $w_j \neq u$: then $\varphi_{w_j \to k_j}(w_i) = \varphi(w_i) = \varphi(u) = \varphi_{w_j \to k_j}(u)$ and $\varphi_{w_i \to k_i}(w_i) \neq \varphi(w_i) = \varphi(u) = \varphi_{w_i \to k_i}(u)$. Theorem 3.1 can be applied with result $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$.

      ii. $[v]_\varphi = \{v, s, \dots\}$. Let w.l.o.g. $w_j \neq s$ (otherwise, if $w_j = s$, then $w_j \neq v$ and a symmetric argument applies). Then $\varphi_{w_i \to k_i}(w_i) = \varphi_{w_i \to k_i}(s)$ but $\varphi_{w_j \to k_j}(w_i) = \varphi(w_i) \neq \varphi(s) = \varphi_{w_j \to k_j}(s)$. In this case, Theorem 3.1 can be applied to obtain $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$.

      iii. $[w_i]_\varphi = \{w_i\}$ and $[v]_\varphi = \{v\}$ and $w_i < v$.

         A. If $w_j = v$. Then neither line 6 (since $|[w_j]_\varphi| = 1$) nor line 8 are executed for w_j and $k = \varphi(w_i)$ (since $w_j \nless w_i$). Thus, $k_j \neq \varphi(w_i)$ must hold. But then $\varphi_{w_i \to k_i}(w_i) = k_i = \varphi(v) = \varphi(w_j) = \varphi_{w_i \to k_i}(w_j)$ and $\varphi_{w_j \to k_j}(w_i) = \varphi(w_i) \neq k_j = \varphi_{w_j \to k_j}(w_j)$. Application of Theorem 3.1 yields $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$.

         B. If $w_j \neq v$. Then $\varphi_{w_i \to k_i}(w_i) = \varphi(v) = \varphi_{w_i \to k_i}(v)$ but $\varphi_{w_j \to k_j}(w_i) = \varphi(w_i) \neq \varphi(v) = \varphi_{w_j \to k_j}(v)$. Again, application of Theorem 3.1 gives us

$$\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\psi).$$

This finishes the proof, since in all cases $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$ holds. □

Lemma 4.2 (Completeness). *For all vertices w and $k \in \{1, \ldots, n\}$, there is $i \in \{1, \ldots, m\}$ such that $\Pi(\varphi_{w \to k}) = \Pi(\varphi_{w_i \to k_i})$ if $\Pi(\varphi_{v \to k}) \neq \Pi(\varphi)$.*

Proof. Let φ be an indexing of V, let w be a vertex and $k \in \{1, \ldots, n\}$. We want to show that if $\Pi(\varphi_{w \to k}) \neq \Pi(\varphi)$, then there is an $i \in \{1, \ldots, m\}$ such that $\Pi(\varphi_{w \to k}) = \Pi(\varphi_{w_i \to k_i})$. First, note that $\varphi(w) \neq k$ holds, since otherwise this would imply $\Pi(\varphi_{w \to k}) = \Pi(\varphi)$. The remainder of this proof works with multiple case distinctions:

1. If $[w]_\varphi = \{w\}$. This directly implies $k = \varphi(v)$ for a $v \in V$, since otherwise that would mean $\Pi(\varphi_{w \to k}) = \Pi(\varphi)$. Thus, for $[v]_\varphi$ there are the following options:
   (a) If $[v]_\varphi = \{v\}$. In the case $w < v$, line 8 enumerates $\varphi_{w \to k}$ directly. Otherwise, if $v < w$, line 8 enumerates $\varphi_{v \to \varphi(w)}$, where $\Pi(\varphi_{v \to \varphi(w)}) = \Pi(\varphi_{w \to k})$.
   (b) If $[v]_\varphi = \{v, s, \ldots\}$. Line 6 in the algorithm directly enumerates $\varphi_{w \to k}$.
2. If $[w]_\varphi = \{w, u, \ldots\}$. Since φ maps w and u to the same index, there is at least one index in $1, \ldots, n$ that is assigned no vertex. But then $\mathcal{N} \neq \emptyset$. Again, there are the following options:
   (a) There is no $v \in V$ such that $k = \varphi(v)$:
      i. If $[w]_\varphi = \{w, v, u, \ldots\}$, then line 11 is executed and there is some $\ell \in \mathcal{N}$ for which $\varphi_{w \to \ell}$ is enumerated. But then $\Pi(\varphi_{w \to k}) = \Pi(\varphi_{w \to \ell})$.
      ii. If $[w]_\varphi = \{w, u\}$.
         A. If $w < u$, then line 11 enumerates $\varphi_{w \to \ell}$ for w and some $\ell \in \mathcal{N}$. But then $\Pi(\varphi_{w \to \ell}) = \Pi(\varphi_{w \to k})$.
         B. If $u < w$, then line 11 enumerates $\varphi_{u \to \ell}$ for u and some $\ell \in \mathcal{N}$. But then again, $\Pi(\varphi_{u \to \ell}) = \Pi(\varphi_{w \to k})$.
   (b) There is $v \in V$ such that $k = \varphi(v)$. Then $k \in \text{image}(\varphi) \setminus \{\varphi(w)\}$ and line 6 is executed. This enumerates $\varphi_{w \to k}$.

This shows that in all cases, there is some $\varphi_{w_i \to k_i}$ that is enumerated which yields the same partition as $\varphi_{v \to k}$. □

Lemma 4.3 (No self-neighbour). $\Pi(\varphi) \neq \Pi(\varphi_{w_i \to k_i})$ *for all $i \in \{1, \ldots, m\}$.*

Proof. $\Pi(\varphi_{v \to k}) = \Pi(\varphi)$ if and only if either $k = \varphi(v)$ or if $[v]_\varphi = \{v\}$ and $k \in \mathcal{N}$. Case distinction yields that both cases never happen for any $\varphi_{w_i \to k_i}$, $i \in \{1, \ldots, m\}$. □

4.2 Move-Enumeration with Random Order

By inspection of algorithm 1, it can be seen that the overall amount of neighbours m of an indexing φ is closely bounded by $|V| \cdot |\Pi(\varphi)|$, since for every vertex v, the amount of k's that are enumerated are at maximum $|\text{image}(\varphi) \setminus \{\varphi(v)\}| + 1 = |\text{image}(\varphi)| - 1 + 1 = |\Pi(\varphi)|$. Since the amount of sets in a partition is bounded by $|V|$, one obtains an amount of neighbours in the order of $O(|V|^2)$. This may be prohibitive if $|V|$ becomes larger, and we are therefore only interested in a smaller part of the neighbourhood of φ, such that the resulting subset is representative for the complete neighbourhood. If the goal is to select a random neighbour from an indexing φ, it should not be required to enumerate all possible neighbours (since this operation is in $O(|V|^2)$) and then having to sample from the resulting set afterwards. Thus, we want to construct a random variable $(\mathbf{w}, \mathbf{k}) \sim \mathcal{U}(\{(w_1, k_1), \dots, (w_m, k_m)\})$ where sampling is not too costly. For simplification, let $S = \{(w_1, k_1), \dots, (w_m, k_m)\}$ be the set of neighbours of φ and define $S_w = \{k : (v, k) \in S, w = v\}$ (i.e., S_w contains exactly the k's that are enumerated together with w on their left side). We can then decompose $\mathcal{U}(\mathbf{w}, \mathbf{k})$ into two parts

$$\mathcal{U}(\mathbf{w}, \mathbf{k}) = \mathcal{Q}(\mathbf{w})\mathcal{P}(\mathbf{k}|\mathbf{w}),$$

where for all $(w, k) \in S$,

$$\mathcal{Q}(\mathbf{w} = w) = \frac{|S_w|}{|S|},$$
$$\mathcal{P}(\mathbf{k} = k|\mathbf{w} = w) = \frac{1}{|S_w|},$$

i.e. $\mathcal{Q}$ models the probability of a vertex w occuring on the left side of a tuple when drawing uniformly from S, and $\mathcal{P}$ models the probability of a k occuring on the right side if w is given (and since every possible k occurs exactly once for a given w, they all have the same probability). Putting it together yields $\mathcal{U}(\mathbf{w} = w, \mathbf{k} = k) = 1/|S|$ for all $(w, k) \in S$, i.e. $\mathcal{U}$ is indeed a uniform probability distribution over S. In order to compute $|S_w|$ for all vertices, we need to be able to count how often a vertex appears on the left side of all tuples. This can be done by looking at algorithm 1, and making case distinctions for $[w]_\varphi$:

1. $[w]_\varphi = \{w, u, v, \dots\}$. Then both line 5 and 9 apply, and $(w, \cdot)$ is enumerated for all $k \in \text{image}(\varphi)\backslash\{\varphi(w)\} \cup \{\ell\}$, where $\ell \in \mathcal{N}$. I.e., we obtain $S_w = \text{image}(\varphi)\backslash\{\varphi(w)\} \cup \{\ell\}$ with $|S_w| = |\text{image}(\varphi)|$.
2. $[w]_\varphi = \{w, u\}$. Then line 5 can be applied and also line 9, but only if $w < u$. This yields $S_w = \text{image}(\varphi)\backslash\{\varphi(w)\} \cup \{\ell\}$ if $w < u$, and $S_w = \text{image}(\varphi)\backslash\{\varphi(w)\}$ if $u < w$. I.e., either $|S_w| = |\text{image}(\varphi)|$ or $|S_w| = |\text{image}(\varphi)| - 1$.
3. $[w]_\varphi = \{w\}$. Line 5 only evaluates to true if the goal index contains at least two different indices, and line 7 only applies if the other vertex is strictly greater than w. Line 9 never applies. Therefore, one obtains $S_w = \{\varphi(v) : v \in V, |[v]_\varphi| \geq 2\} \cup \{\varphi(v) : v \in V, [v]_\varphi = \{v\}, w < v\}$ and thus $|S_w| = |\{\varphi(v) : v \in V, |[v]_\varphi| \geq 2\}| + |\{\varphi(v) : v \in V, [v]_\varphi = \{v\}, w < v\}|$. The first summand can be computed relatively easy and is the same for all vertices, but the second might cause problems, since its computation depends on w. Fortunately, this can be solved: let $K = \{\varphi(v) : v \in V, [v]_\varphi = \{v\}\}$ be the set that contains indices which only have one vertex, and let $K_w = \{\varphi(v) \in K : w < v\}$ be the subset of K that contains only indices of vertices which are strictly greater than w (note that $K_w = \{\varphi(v) : v \in V, [v]_\varphi = \{v\}, w < v\}$). Assume for $K = \{\varphi(v_1), \dots, \varphi(v_{|K|})\}$, we can write down the order of the elements as $v_1 < \dots < v_i < v_{i+1} < \dots < v_{|K|}$. Thus, $K_{v_i} = \{\varphi(v_{i+1}), \dots, \varphi(v_{|K|})\}$ and therefore $|K_{v_i}| = |K| - i$. One can then pre-compute $|K_{v_i}|$ for every vertex v_i with $[v_i]_\varphi = \{v_i\}$ by iterating over the linear order for $i = 1, \dots, |K|$.

Putting it together, we obtain

$$|S_w| = \begin{cases} |\mathrm{image}(\varphi)| & [w]_\varphi = \{w, u, v, \dots\} \\ |\mathrm{image}(\varphi)| & [w]_\varphi = \{w, u\} \text{ and } w < u \\ |\mathrm{image}(\varphi)| - 1 & [w]_\varphi = \{w, u\} \text{ and } u < w \\ |\{\varphi(v) : v \in V, |[v]_\varphi| \geq 2\}| + |K_w| & [w]_\varphi = \{w\} \end{cases} \tag{4.2}$$

and hence, algorithm 2.

Algorithm 2: Random Move-Enumeration

```
Input: Set of vertices $V$ with indexing $\varphi$, number of neighbours $N$
Result: Uniformly random moves $(w_1, k_1), \dots, (w_N, k_N)$
1  Let $<$ be a linear order on $V$
2  Compute $|\mathrm{image}(\varphi)|$ and $|\{\varphi(v) : v \in V, |[v]_\varphi| \geq 2\}|$
3  Compute $K = \{\varphi(v) : v \in V, [v]_\varphi = \{v\}\} = \{\varphi(v_1), \dots, \varphi(v_{|K|})\}, v_1 < \dots < v_{|K|}$
4  For each vertex $w$, compute $|S_w|$ from formula (4.2) and results from line 2 and 3
5  Compute $|S| = \sum_{v \in V} |S_v|$
6  for $i = 1, \dots, N$ do
7      Sample $w$ with probability $|S_w|/|S|$ from $V$
8      Generate $S_w$ through enumeration of all possible $k$'s as in line 4 - 11 in algorithm 1
9      Sample $k$ with probability $1/|S_w|$ from $S_w$
10     enumerate $(w, k)$
```

As already stated, enumerating the complete neighbourhood and then sampling N neighbours afterwards would be bounded by $O(|V| \cdot |\Pi(\varphi)| + N) = O(|V|^2 + N)$ steps. Let us now check that the time of algorithm 2 is indeed better: lines 1 to 5 each take approximately $|V|$ steps, which yields a bound of $O(|V|)$. Line 7 and 8 take overall $|V|$ and $|\Pi(\varphi)|$ steps[4], and 9 can be executed in constant time[5]. Therefore, we obtain a run-time of $O(|V| + N(|V| + |\Pi(\varphi)|)) = O(N \cdot |V|)$. Thus, if $N \ll |V|$, we are able to obtain a much faster way of generating neighbours for a given indexing.

5 Confronting the Objective Function

Since we are only interested in some minimizer Π^* of the original problem, it does not matter which function we minimize, as long as the set of minimizers stays the same. Therefore, we can apply any strongly monotonic growing function to the objective, and, for example, multiply by some positive constant. We are then able to obtain

$$\Pi^* = \arg\min_{\Pi \in P_V} \sum\nolimits_{T \in \binom{V}{3}} \ell(T, \Pi) \tag{5.1}$$

$$= \arg\min_{\Pi \in P_V} \frac{1}{|\binom{V}{3}|} \sum\nolimits_{T \in \binom{V}{3}} \ell(T, \Pi) \tag{5.2}$$

$$= \arg\min_{\Pi \in P_V} \mathbb{E}_{\mathbf{T} \sim \mathcal{U}(\binom{V}{3})} [\ell(\mathbf{T}, \Pi)] \tag{5.3}$$

[4]The execution time of line 7 is linear in $|V|$ since we are sampling from a discrete probability distribution that is not uniform.

[5]Since we are sampling with uniform probability from S_w.

where $\mathcal{U}$ is the uniform distribution. Equality (5.1) is just the definition of Π^*, (5.2) is the same function multiplied by a positive value and the last equality (5.3) is just the definition of the expected value. This perspective allows to approximate the objective value to any degree by uniformly sampling a fixed number of 3-ary subsets from V and computing the sample mean.

5.1 Computing Value Improvements

In many cases, one wants to compute the change in the objective function when considering two different partitions $\Pi, \Gamma \in P_V$. The difference is then given by

$$\begin{aligned}\mathbb{E}_{\mathbf{T}\sim\mathcal{U}(\binom{V}{3})}[\ell(\mathbf{T},\Pi)] - \mathbb{E}_{\mathbf{T}\sim\mathcal{U}(\binom{V}{3})}[\ell(\mathbf{T},\Gamma)] &= \mathbb{E}_{\mathbf{T}\sim\mathcal{U}(\binom{V}{3})}[\ell(\mathbf{T},\Pi) - \ell(\mathbf{T},\Gamma)] \\ &= \mathbb{E}_{\mathbf{T}\sim\mathcal{U}(\binom{V}{3})}[\delta(\mathbf{T},\Pi,\Gamma)]\end{aligned}$$

which can possibly be simplified, dependent on the shape of Γ with respect to Π and vice versa. For example, if we consider an indexing φ of V with $\Pi = \Pi(\varphi)$ and $\Gamma = \Pi(\varphi_{v\to k})$, i.e. Γ is the result of a move-operation on Π, then the above can be simplified to

$$\mathbb{E}_{\{\mathbf{u},\mathbf{w}\}\sim\mathcal{U}(\binom{V\setminus\{v\}}{2})}[\delta(\{\mathbf{u}, v, \mathbf{w}\},\Pi,\Gamma)] = J(\Pi,\Gamma),$$

since $\ell(T,\Pi) = \ell(T,\Gamma)$ for all $T \in \binom{V\setminus\{v\}}{3}$. This can be shown as follows. Take $u, w \in V\setminus\{v\}$. Then

$$\begin{aligned}[u]_{\Pi(\varphi)} = [w]_{\Pi(\varphi)} \quad &\text{iff} \quad \varphi(u) = \varphi(w) && (5.4)\\ &\text{iff} \quad \varphi_{v\to k}(u) = \varphi_{v\to k}(w) && (5.5)\\ &\text{iff} \quad [u]_{\Pi(\varphi_{v\to k})} = [w]_{\Pi(\varphi_{v\to k})} && (5.6)\end{aligned}$$

Equivalences (5.4) and (5.6) follow from Lemma 3.2, and (5.5) holds since $u \neq v \neq w$. Therefore, if T is a 3-ary subset of V that does not contain v, it can be seen that the conditions that make $\ell(T,\Pi(\varphi))$ or $\ell(T,\Pi(\varphi_{v\to k}))$ take certain values are exactly the same, which implies equality. Overall, this allows for a computation time bounded by $|V|^2$ if the expected value is computed explicitly.

6 Algorithms

We are now ready to present a local search algorithm that greedily improves a given partition (algorithm 3). This algorithm takes as input a set of vertices, an indexing that is used as the initial value and a stopping criteria that depends on the number of iterations and the current indexing. In every iteration, it generates the neighbourhood of the current indexing through application of algorithm 1 (line 3). Afterwards, in line 4, the neighbour with the best improvement is selected. If there is no neighbour that yields any better value (i.e. the difference computed in line 4 is negative or zero), a local minima was found and the current indexing is returned (line 6). Otherwise, the algorithm just continues (line 7).
Considering an indexing φ, one can see that the amount of possible neighbours is bounded by $O(|V| \cdot |\Pi(\varphi)|)$ – this becomes clear by introspection of algorithm 1, since for every vertex, there are at maximum $|\mathrm{image}(\varphi)| = |\Pi(\varphi)|$ candidates for k. Fully computing the improvement of a neighbour in line 4 is closely bounded by $O(|V|^2)$ steps, since one has to compute almost all possible 2-ary combinations of vertices. Combining this with the size

of the neighbourhood, one obtains an overall upper bound of $O(|V|^3 \cdot |\Pi(\varphi)|)$ required steps for computing the best neighbour. The remainder, i.e. line 6 and 7, can be computed in constant time if one does not recompute the improvement of $\varphi_{v^* \to k^*}$ over φ or copies the whole indexing. Since the amount of sets in a partition is bounded by $|V|$, we obtain an upper bound of $O(|V|^4)$ steps per iteration. However, if we assume that the maximal number of iterations is bounded by a constant, this complexity propagates to the complete algorithm[6].

Algorithm 3: Greedy-Search

```
Input: Set of vertices $V$ with indexing $\varphi$ and stopping criteria $\mathrm{stop} : \mathbb{N} \times [n]^V \to \{0,1\}$.
Result: Better indexing $\psi$
1 Let $i := 1$
2 while not $\mathrm{stop}(i, \varphi)$ do
3     Let $(v_1, k_1), \dots, (v_m, k_m)$ be the output generated by Algorithm 1 on input $V$ and $\varphi$
4     Compute $(v^*, k^*) := \arg\max_{(v_i, k_i)} J(\Pi(\varphi), \Pi(\varphi_{v_i \to k_i}))$
5     if $J(\Pi(\varphi), \Pi(\varphi_{v^* \to k^*})) \leq 0$ then
6         return $\varphi$
7     Set $\varphi := \varphi_{v^* \to k^*}$ and $i := i + 1$
8 return $\varphi$
```

A second variant of the greedy search algorithm makes use of sampling. Algorithm 4 does not consider the complete neighbourhood of any indexing, but rather randomly selects a given number of N neighbours (line 3). For each of these neighbours, a fixed number of M random vertices is selected (line 4), which are used to compute the sample mean in line 5. The only real conceptual difference to algorithm 3 is that instead of returning the current indexing if no neighbour yields an improvement, algorithm 4 just continues (line 6 and 7). This is because of the variance that is induced when sampling vertices or neighbours, i.e.: the selected "best" neighbour (line 5) might not actually be worse than the current solution or the actual best neighbour was not sampled from the neighbourhood (line 3).

Algorithm 4: Greedy-Search with Sampling

```
Input: Set of vertices $V$ with indexing $\varphi$, stopping criteria $\mathrm{stop} : \mathbb{N} \times [n]^V \to \{0,1\}$,
       neighbourhood sample size N, objective sample size M
Result: Better indexing $\psi$
1 Let $i := 1$
2 while not $\mathrm{stop}(i, \varphi)$ do
3     Let $(v_1, k_1), \dots, (v_N, k_N)$ be the output of Algorithm 2 on input $V$, $\varphi$ and $N$
4     Sample $\{u_{i,1}, w_{i,1}\}, \dots, \{u_{i,M}, w_{i,M}\}$ from $\binom{V \setminus \{v_i\}}{2}$ for all $i \in \{1, \dots, N\}$
5     Compute $(v^*, k^*) := \arg\max_{(v_i, v_i)} \frac{1}{M} \sum_{j=1}^{M} \delta(\{u_{i,j}, w_{i,j}, v_i\}, \Pi(\varphi_i), \Pi(\varphi_{v_i \to k_i}))$
6     if $\varphi_{v^* \to k^*}$ is an improvement over $\varphi$ then
7         Set $\varphi := \varphi_{v^* \to k^*}$
8     Set $i := i + 1$
9 return $\varphi$
```

[6] Although it would make sense to choose the stopping criteria dependent on the number of vertices, since in each iteration, there is only one vertex that is moved, and the overall number of vertices might be too high to reach an optimal solution in time.

We now want to analyze the complexity of an iteration in algorithm 4. As already discussed in section 4.2, line 3 takes $O(|V| \cdot N)$ steps. By sampling $N \cdot M$ values in line 4, overall $O(N \cdot M)$ steps are required. Computing the sample mean of the costs for a pair (v_i, k_i) in line 5 take $O(M)$ steps, and since these costs are computed for N different pairs, this yields a bound of $O(N \cdot M)$ steps. The remainder, i.e. line 7 and 8, can be done in constant time. This leaves us with an overall bound of $O(|V| \cdot N + N \cdot M) = O(N \cdot (|V| + M))$ steps per iteration.

7 Experiments

We evaluate the proposed algorithms on the following scenario: a collection of 3d-points $P = \{p_1, \dots, p_s\} \subseteq \mathbb{R}^3$ is sampled from a number of t planes which contain the coordinate origin. We wish to partition the resulting point cloud in such a way that makes the underlying set of planes obvious: i.e., all points that were sampled from a given plane are part of the same partition, distinct from the partitions of the other planes. Figure 2 shows some examples on possible partitions. The points are generated as follows:

1. Every plane is defined by a normal vector and an orientation, i.e. we have vectors $\vec{n}_1, \dots, \vec{n}_t \in \mathbb{R}^3$ and orientations $\vec{o}_1, \dots, \vec{o}_t \in \mathbb{R}^3$. For each plane, we draw λ_i, λ_i' from $\mathcal{U}([-\pi, \pi))$, and x_i, x_i' from $\mathcal{U}([0, 1))$ and determine $\varphi_i = \arccos(2x_i - 1)$, $\varphi_i' = \arccos(2x_i' - 1)$[7]. The vectors are computed by

$$\vec{n}_i = \begin{bmatrix} \sin(\varphi_i)\cos(\lambda_i) \\ \sin(\varphi_i)\sin(\lambda_i) \\ \cos(\varphi_i) \end{bmatrix}, \quad \vec{o}_i = \operatorname{norm}\left(\begin{bmatrix} \sin(\varphi_i')\cos(\lambda_i') \\ \sin(\varphi_i')\sin(\lambda_i') \\ \cos(\varphi_i') \end{bmatrix} \times \vec{n}_i \right),$$

   where $\operatorname{norm}(\vec{h}) = \frac{1}{||\vec{h}||}\vec{h}$ ($\vec{n}_i$ is already normalized, since it describes a point on the unit-sphere).

2. For every point p_i, three components are sampled: the 2d-position on the plane x_i, y_i from $\mathcal{U}([-1, 1])$, and a noise-component z_i from $\mathcal{U}([-\epsilon, \epsilon])$ (where ϵ indicates the level of noise). The point is then transformed into plane-space, i.e.

$$p_i = x_i\vec{o}_i + y_i\operatorname{norm}(\vec{o}_i \times \vec{n}_i) + z_i\vec{n}_i$$

For three points $u, v, w \in P$, we define the following cost-structure

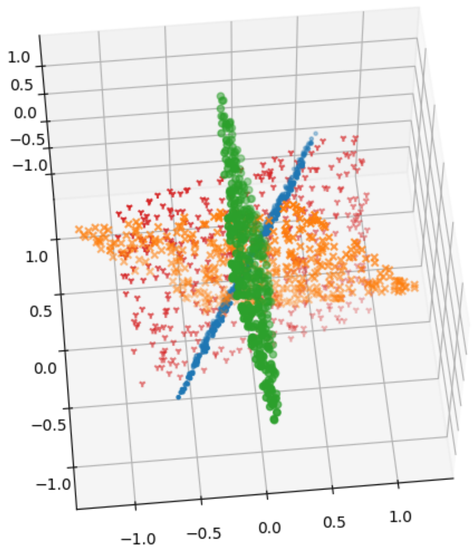
$$c(\{u, v, w\}) = 0, \quad c'(\{u, v, w\}) = \mathrm{d}(\{u, v, w\}, (0, 0, 0)) - \tau,$$

where $\mathrm{d}(\{u, v, w\}, p)$ is the distance between the plane that contains u, v and w and p ("$\cdot$" is the dot-product in this case):

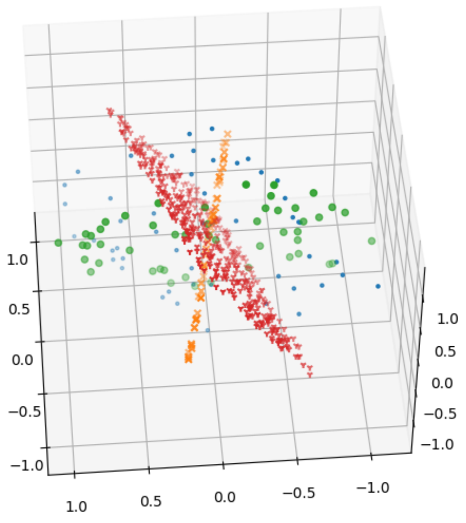
$$\mathrm{d}(\{u, v, w\}, p) = \frac{\vec{n} \cdot (u - p)}{||\vec{n}||}, \quad \vec{n} = (v - u) \times (w - u),$$

and τ is some small value. Therefore, if the distance from the plane spanning over u, v and w to the origin ist smaller than τ, it is considered "good" when u, v and w are part of the same partition and if the distance is large, it is considered "bad".

[7]Source: Jason Davies, https://www.jasondavies.com/maps/random-points/. Accessed on December 6th, 2020.



(a) Point cloud consisting of four planes with 400 points each and noise $\epsilon = 0.01$.



(b) Point cloud consisting of four planes, of which three have 50 points and one has 400 points with noise $\epsilon = 0.01$.

Figure 2: Two possible point clouds.

dataset	planes × number
0	3×30
1	3×80
2	3×150
3	4×250
4	5×300

Table 1: Generated datasets and executed runs. Every set contains a number of planes with an equal amount of points. Noise level is $\epsilon = 0.01$.

run	dataset	**ip**	N	M	**or** [s]	**iter**
0-S-S	0	1	40	1000	3.00	153
0-E-S	0	1	–	1000	37.57	109
0-S-E	0	1	40	–	4.73	148
0-E-E	0	1	–	–	23.80	100
1-S-S	1	1	40	5000	18.31	518
1-E-S	1	1	–	5000	1348.06	511
1-S-E	1	1	40	–	62.00	439
1-E-E	1	1	–	–	511.49	131
2-S-S	2	1	50	5000	43.98	958
3-S-S	3	1	50	10000	322.22	4607
4-S-S	4	1	50	15000	1280.99	10694
2-S-SL	2	1	50	500	37.80	1769
2-SL-S	2	1	5	5000	78.19	10480
2-S-S-IP10	2	10	50	5000	62.54	1309
2-S-S-IP50	2	50	50	5000	74.12	1582
2-S-S-IP100	2	100	50	5000	88.77	1937

Table 2: Every run with the corresponding dataset it was executed on, the number of **i**nitial **p**artitions, the number of neighbours N and samples M that are selected per iteration, the **o**verall **r**untime and number of **iter**ations. If entries for N or M are "–", then this means that the complete neighbourhood was considered, or the reduced costs where computed explicitly. For each run, $\tau = 0.1$ was selected.

The algorithms are evaluated in the four following settings. First, the deterministic greedy search is compared with randomized versions, in which only a certain subset of neighbours is selected and/or the reduced cost is only approximated. This is done on the two datasets 0 and 1, in which dataset 0 contains 3×30 points, and dataset 1 overall 3×80 points. In the second setting, problems with larger input-sizes are considered and solved by the randomized greedy search algorithm (dataset 2,3 and 4 with 3×150, 4×250 and 5×300 points respectively). The penultimate setting examines the impact of low neighbourhood- and sampling sizes regarding the cost function. Finally, the randomized greedy search algorithm is initialized with multiple randomly assigned partitions. The results are shown in figures 3, 4, 5, 6 and 7. The upper rows show how much the costs are reduced in each iteration, the middle rows contain the used time per iteration – the cumulative time to compute the reduced cost for each neighbour in orange, and the time required to compute the neighbourhood in blue – and the last rows show the number of partitions per iteration.

Setting 1. The results are depicted in figures 3 and 4. When the complete neighbourhood is considered, as for run 0-E-S, 0-E-E, 1-E-S and 1-E-E, the used time per iteration is clearly dependent on the number of partitions. This is obvious from analysis of algorithm 1, since the number of neighbours grows in size $O(|V| \cdot \Pi(\varphi))$. Considering the complete neighbourhood has probably the largest impact on the overall runtime, as can be seen in table 2.

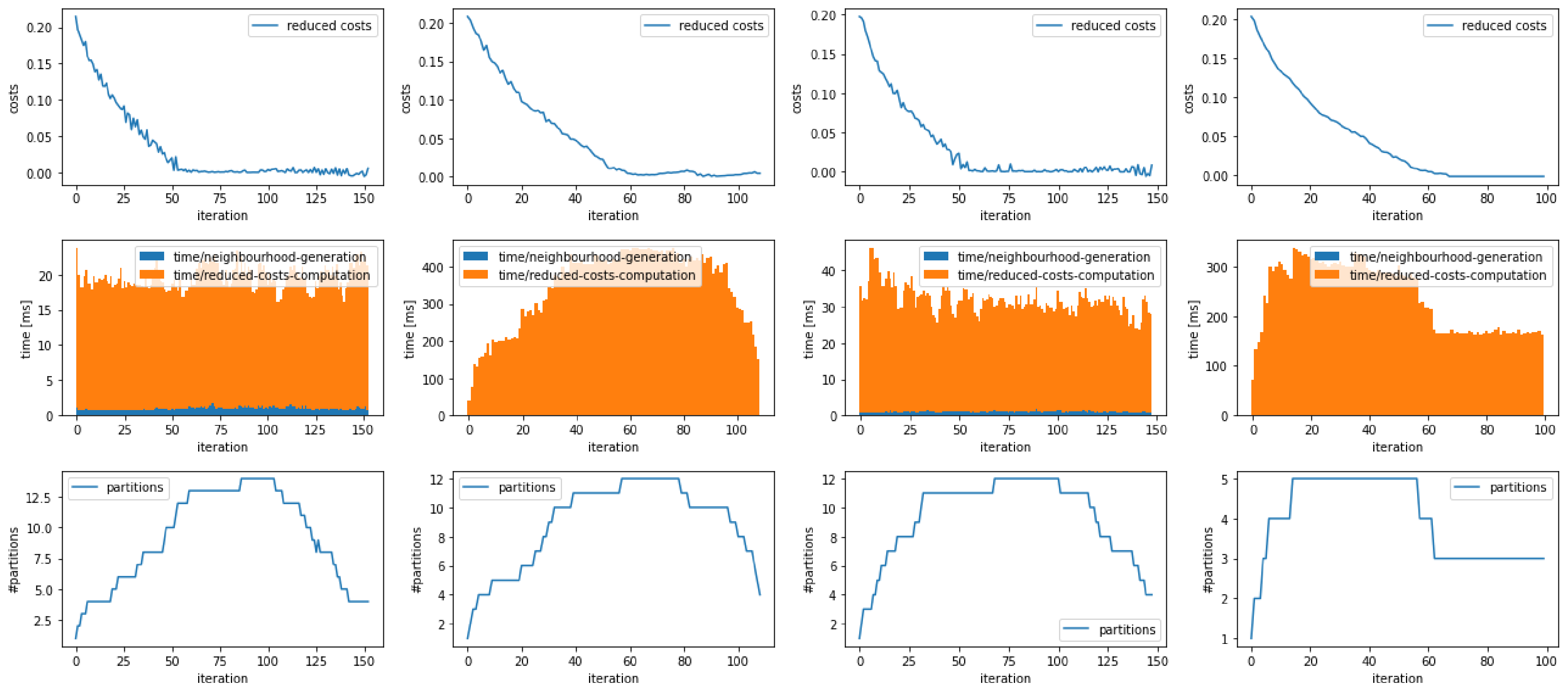


Figure 3: From left to right: run 0-S-S, 0-E-S, 0-S-E, 0-E-E.

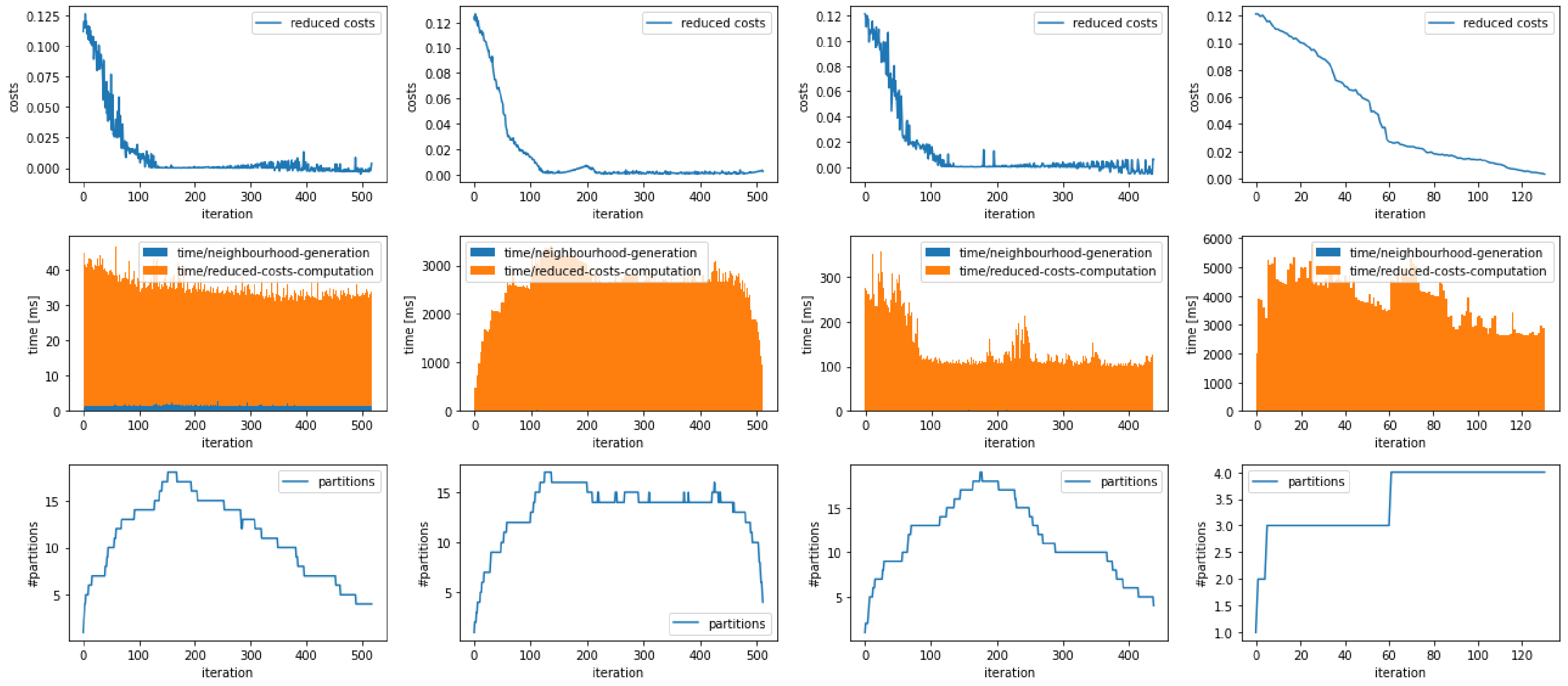


Figure 4: From left to right: run 1-S-S, 1-E-S, 1-S-E, 1-E-E.

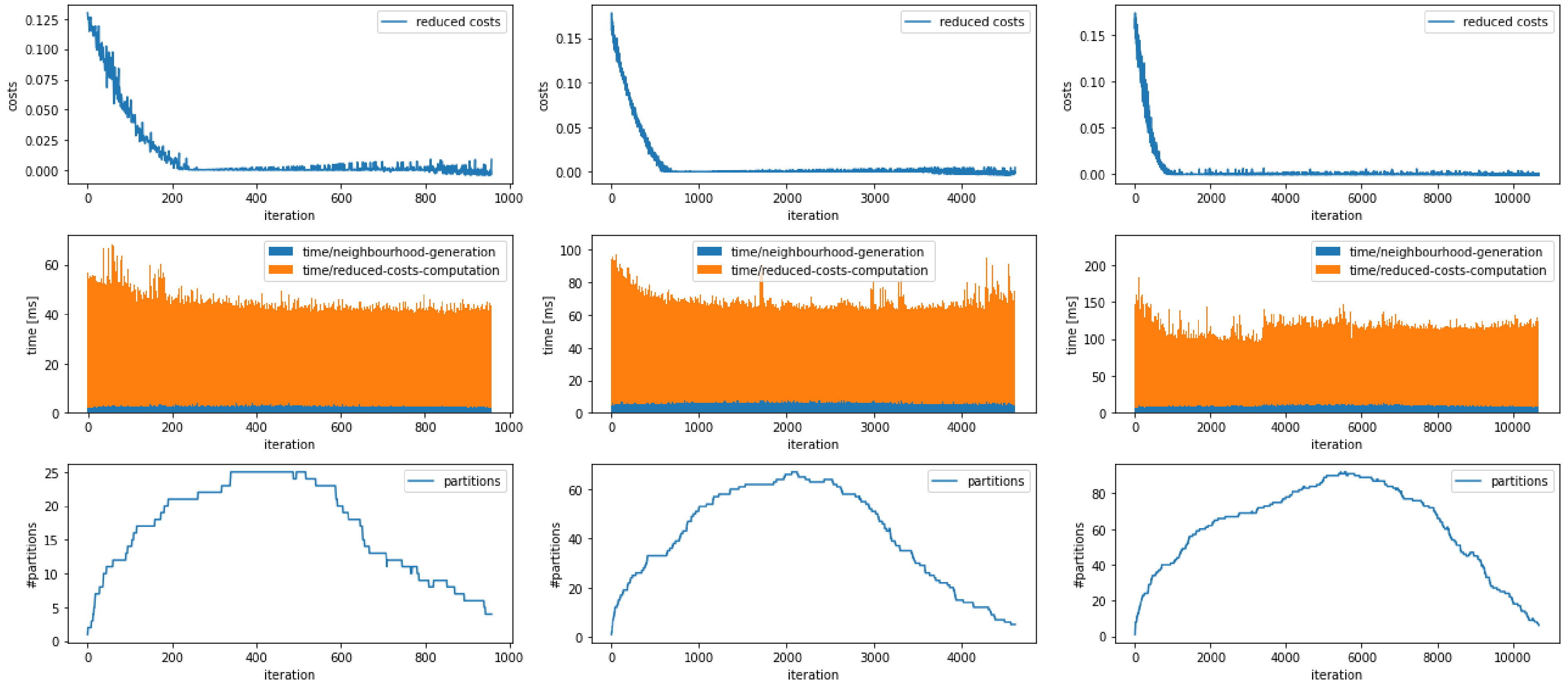


Figure 5: From left to right: run 2-S-S, 3-S-S, 4-S-S.

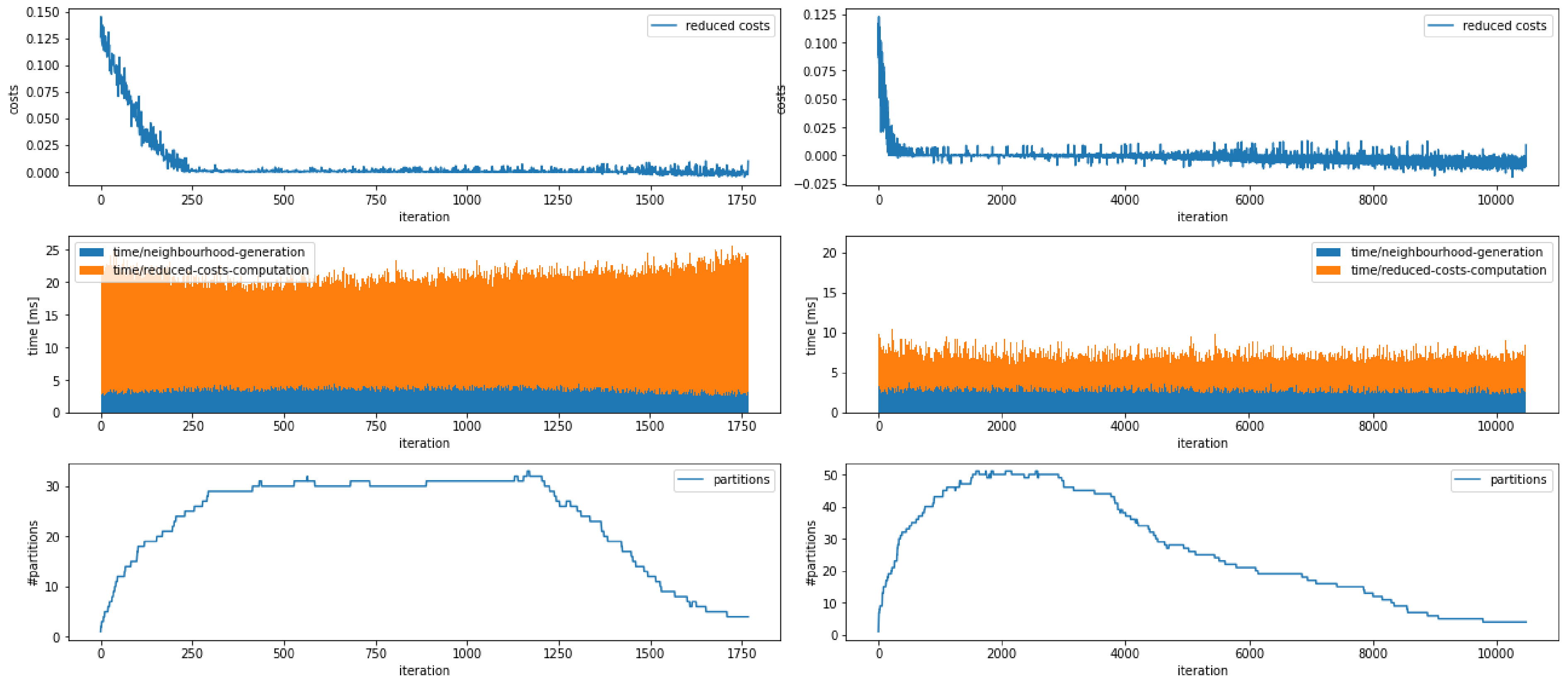


Figure 6: From left to right: run 2-S-SL, 2-SL-S.

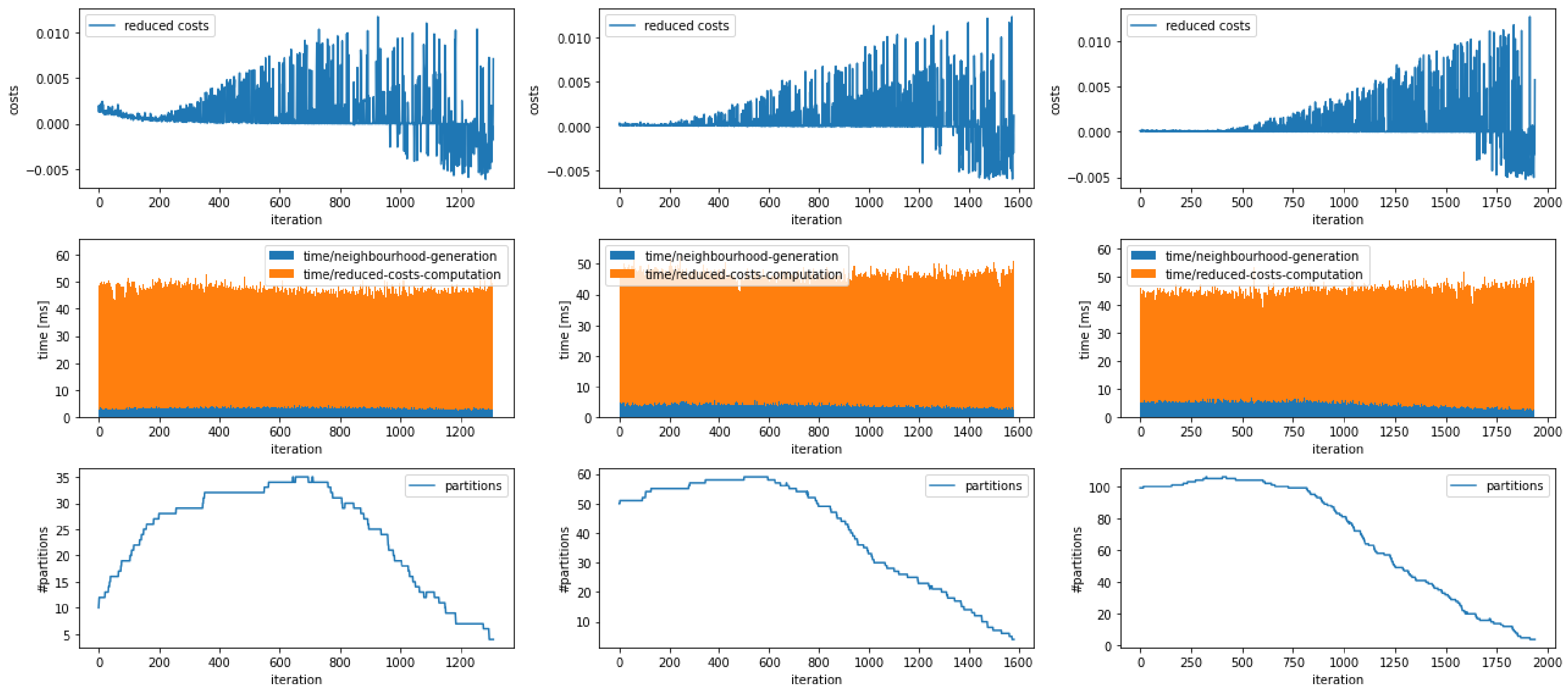


Figure 7: From left to right: run 2-S-S-IP10, 2-S-S-IP50, 2-S-S-IP100.

References

[1] David H Wolpert, William G Macready, et al. No free lunch theorems for search. Technical report, Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.