A Local Search Algorithm for Cubic Clustering

Hans Harder, TU Dresden

Contents

1 Notation **2**

2 Introduction **2**

3 Indexings as Proxies for Partitions **3**

4 Move-Operation on Indexings **5**
4.1 Move-Enumeration 5
4.2 Move-Enumeration with Random Order 6

5 Computing the Objective Function **9**
5.1 Computing Value Improvements 9

6 Algorithms **10**

7 Experiments **11**

8 Conclusion and Possible Improvements **21**

A Proofs for Section 3 (Indexings as Proxies for Partitions) **22**

B Proofs for Section 4 (Move-Operation on Indexings) **23**

1 Notation

Let A be some set. We will denote the set $\{\{a_1, \dots, a_k\} \subseteq A : a_1, \dots, a_k \text{ distinct}\}$, 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 $\{x_1, \dots, x_n\}$ (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., [2]) 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)
- ...computation (or estimation) of the given cost function and cost-improvements.

The problem is given as follows. Let $V = \{v_1, \dots, v_n\}$ 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 a cost-structure on 3-ary subsets of V. For a 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

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

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 *Indexings as Proxies for Partitions*, we will discuss an efficient way of representing partitions, e.g. with respect to their space requirements. Afterwards, in section *Move-Operation on Indexings*, 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 *Computing the Objective Function* 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 *Algorithms* contains a theoretic discussion on how the proposed settings influence the runtime of a simple greedy search algorithm, which will be evaluated in section *Experiments*. Proofs for Lemmas and Theorems that are shown in this work can be found in the appendix.

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 $O(|V|)$.

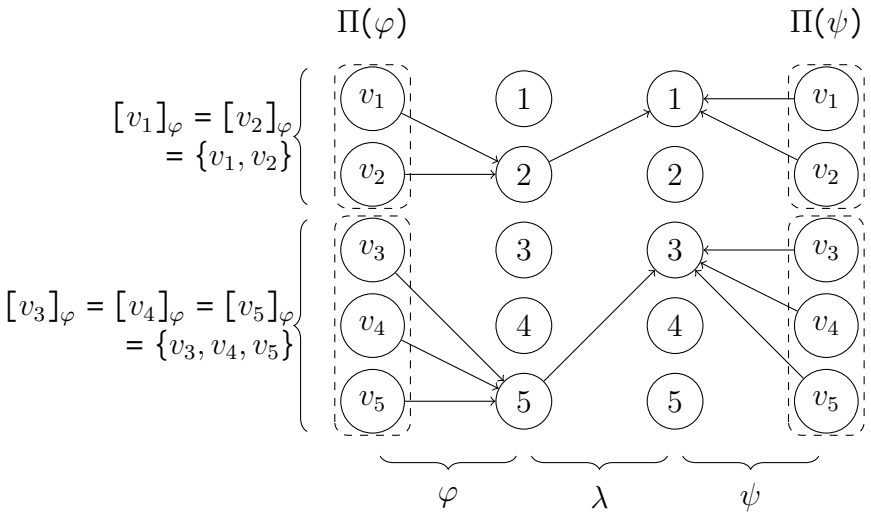


Figure 1: Illustration of two indexings φ, ψ with corresponding partitions and proof of equality through part (b) of Theorem 3.3.

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 indexing that yields that very partition (Lemma 3.1). And two, we obtain a criterion 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.*

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 $\Pi(\varphi)$.*

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 3.3.

Theorem 3.3. *If φ, ψ are two indexings of V, then the following statements are equivalent:*

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

(b) there is a bijection $\lambda : \mathrm{image}(\varphi) \to \mathrm{image}(\psi)$[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* $\psi(w) = \psi(v)$

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 3.3 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 Theorem 3.3 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 3.3.

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

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

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

Algorithm 1: Move-Enumeration (ME)

Input: Set of vertices V with indexing φ
Result: Sequence of moves $(w_1, k_1), \ldots, (w_m, k_m)$

```
1  Let < be some linear order on V
2  Let 𝒩 := {1, ..., n}\image(φ)
3  forall vertices w ∈ V do
4      forall φ(v) ∈ image(φ)\{φ(w)} do
5          if [w]_φ = {w, u, ...} or [v]_φ = {v, s, ...} then
6              enumerate (w, φ(v))
7          else if [w]_φ = {w} and [v]_φ = {v} and w < v then
8              enumerate (w, φ(v))
9      if [w]_φ = {w, u, v, ...} or ([w]_φ = {w, u} and w < u) then
10         Let k ∈ 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 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 *ME* 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 $(w_1, k_1), \ldots, (w_m, k_m)$ is a sequence generated by algorithm *ME* on input V and φ, Lemma 4.1, 4.2 and 4.3 hold.

Lemma 4.1 (Pairwise Distinctiveness). *If* $i \neq j$, *then* $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$.

Lemma 4.2 (Completeness). *If* $w \in V$ *and* $1 \leq k \leq n$ *and* $\Pi(\varphi_{w \to k}) \neq \Pi(\varphi)$, *then there is* $1 \leq i \leq m$ *with* $\Pi(\varphi_{w \to k}) = \Pi(\varphi_{w_i \to k_i})$.

Lemma 4.3 (No self-neighbour). *If* $1 \leq i \leq m$, *then* $\Pi(\varphi) \neq \Pi(\varphi_{w_i \to k_i})$.

4.2 Move-Enumeration with Random Order

By inspection of algorithm *ME*, 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 number of k's that are enumerated is at maximum $|\mathrm{image}(\varphi)\backslash\{\varphi(v)\}| + 1 = |\mathrm{image}(\varphi)| - 1 + 1 = |\Pi(\varphi)|$. 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

[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.

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. Therefore, we are interested in sampling a random subset of neighbours with predefined size. 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. By Lemmas 4.2, 4.1 and 4.3, it is easy to see that there is a 1:1 correspondence between the neighbours of a partition and the elements $(w_1, k_1), \dots, (w_m, k_m)$ that are enumerated through algorithm *ME*. Thus, if we want to select a random neighbour, we can simply pick a random tuple (w, k) from the sequence. In other words, we want to construct a random variable $(\mathbf{w}, \mathbf{k}) \sim \mathcal{U}(\{(w_1, k_1), \dots, (w_m, k_m)\})$ such that sampling is not too costly. For simplification, let $S^\varphi = \{(w_1, k_1), \dots, (w_m, k_m)\}$ be the set of neighbours of φ and define $S^\varphi_w = \{k : (v, k) \in S^\varphi, w = v\}$ (i.e., S^φ_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 $(w, k) \in S^\varphi$,

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

i.e. $\mathcal{Q}$ models the probability of a vertex w occuring on the left side of a tuple when drawing uniformly from S^φ, 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 $\mathcal{U}(\mathbf{w} = w, \mathbf{k} = k) = 1/|S^\varphi|$ for all $(w, k) \in S^\varphi$, i.e. $\mathcal{U}$ is indeed a uniform probability distribution over S^φ. In order to compute $|S^\varphi_w|$ 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 *ME*, 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 $S^\varphi_w = \text{image}(\varphi)\backslash\{\varphi(w)\} \cup \{\ell\}$ with $|S^\varphi_w| = |\text{image}(\varphi)|$.

2. $[w]_\varphi = \{w, u\}$. Then line 5 can be applied and also line 9, but only if $w < u$. This yields $S^\varphi_w = \text{image}(\varphi)\backslash\{\varphi(w)\} \cup \{\ell\}$ if $w < u$, and $S^\varphi_w = \text{image}(\varphi)\backslash\{\varphi(w)\}$ if $u < w$. I.e., either $|S^\varphi_w| = |\text{image}(\varphi)|$ or $|S^\varphi_w| = |\text{image}(\varphi)| - 1$.

3. $[w]_\varphi = \{w\}$. Line 5 only evaluates to true if the target index contains at least two different vertices, and line 7 only applies if the other vertex is strictly greater than w. Line 9 never applies. Therefore, one obtains $S^\varphi_w = \{\varphi(v) : v \in V, |[v]_\varphi| \geq 2\} \cup \{\varphi(v) : v \in V, [v]_\varphi = \{v\}, w < v\}$ and thus $|S^\varphi_w| = |\{\varphi(v) : v \in V, |[v]_\varphi| \geq 2\}| + |\{\varphi(v) : v \in V, [v]_\varphi = \{v\}, w < v\}|$. 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 $v_1 < \cdots < v_i < v_{i+1} < \cdots < v_{|K|}$. 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

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

and hence, algorithm *RME*.

Algorithm 2: Random Move-Enumeration (RME)

Input: Set of vertices V with indexing φ, 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 $K = \{\varphi(v) : v \in V, [v]_\varphi = \{v\}\} = \{\varphi(v_1), \dots, \varphi(v_{|K|})\}$, $v_1 < \cdots < v_{|K|}$ and $|K_{v_1}|, \dots, |K_{v_{|K|}}|$
4 For each vertex w, compute $|S_w^\varphi|$ from formula (4.2) and results from line 2 and 3
5 Compute $|S^\varphi| = \sum_{v \in V} |S_v^\varphi|$
6 **for** $i = 1, \dots, N$ **do**
7 Sample w with probability $|S_w^\varphi|/|S^\varphi|$ from V
8 Generate S_w^φ through enumeration of all possible k's as in line 5 - 11 in algorithm *ME* with the same linear order
9 Sample k with probability $1/|S_w^\varphi|$ from S_w^φ
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 worst case time of algorithm *RME* 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 faster way of generating neighbours for a given indexing.

[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^φ.

5 Computing 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

$$\begin{aligned}\Pi^* &= \operatorname*{arg\,min}_{\Pi \in P_V} \sum\nolimits_{T \in \binom{V}{3}} \ell(T, \Pi) && (5.1)\\ &= \operatorname*{arg\,min}_{\Pi \in P_V} \frac{1}{\left|\binom{V}{3}\right|} \sum\nolimits_{T \in \binom{V}{3}} \ell(T, \Pi) && (5.2)\\ &= \operatorname*{arg\,min}_{\Pi \in P_V} \mathbb{E}_{\mathbf{T} \sim \mathcal{U}\left(\binom{V}{3}\right)} \left[\ell(\mathbf{T}, \Pi)\right] && (5.3)\end{aligned}$$

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

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

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

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

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 J is computed explicitly.

6 Algorithms

We are now ready to present a local search algorithm that greedily improves a given partition (algorithm *GS*). This algorithm takes as input a set of vertices, an indexing that is used as the initial value and a stopping criterion 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 *ME* (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 *ME*, 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 (GS)

Input: Set of vertices V with indexing φ and stopping criterion
$\mathrm{stop} : \mathbb{N} \times [n]^V \to \{0, 1\}$.
Result: Better indexing ψ

```
1 Let i := 1
2 while not stop(i, φ) do
3 |   Let (v_1, k_1), ..., (v_m, k_m) be the output from algorithm ME on input V, φ
4 |   (v*, k*) := arg max_{(v_i,k_i)} J(Π(φ), Π(φ_{v_i→k_i}))
5 |   if J(Π(φ), Π(φ_{v*→k*})) ≤ 0 then
6 |   |   return φ
7 |   Set φ := φ_{v*→k*} and i := i + 1
8 return φ
```

A second variant of the greedy search algorithm makes use of sampling. Algorithm *GSS* 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

[6] Although it would make sense to choose the stopping criterion 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.

the sample mean in line 5. Another conceptual difference to algorithm *GS* is that instead of returning the current indexing if no neighbour yields an improvement, algorithm *GSS* 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 (GSS)

Input: Set of vertices V with indexing φ, stopping criterion $\mathrm{stop} : \mathbb{N} \times [n]^V \to \{0,1\}$, neighbourhood sample size N, objective sample size M

Result: Better indexing ψ

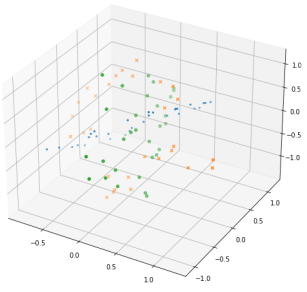
1 Let $i := 1$
2 **while** *not* $\mathrm{stop}(i, \varphi)$ **do**
3 Let $(v_1, k_1), \ldots, (v_N, k_N)$ be the output of Algorithm *RME* on input V, φ, N
4 Sample $\{u_{i,1}, w_{i,1}\}, \ldots, \{u_{i,M}, w_{i,M}\}$ from $\binom{V \setminus \{v_i\}}{2}$ for all $i \in \{1, \ldots, N\}$
5 $(v^*, k^*) := \arg\max_{(v_i, v_i)} \frac{1}{M} \sum_{j=1}^{M} \delta(\{u_{i,j}, w_{i,j}, v_i\}, \Pi(\varphi_i), \Pi(\varphi_{v_i \to k_i}))$
6 **if** $\varphi_{v^* \to k^*}$ *is an improvement over* φ **then**
7 Set $\varphi := \varphi_{v^* \to k^*}$
8 Set $i := i + 1$
9 **return** φ

We now want to analyze the complexity of an iteration in algorithm *GSS*. 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.

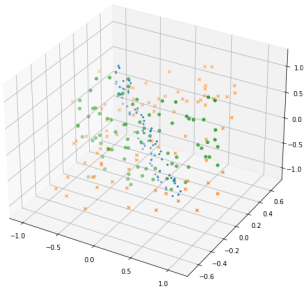
7 Experiments

We evaluate the proposed algorithms on the following scenario: a collection of 3d-points $P = \{p_1, \ldots, p_s\} \subseteq \mathbb{R}^3$ 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 set in the partition, distinct from the sets of the other planes. The points are generated as follows. First, every plane is defined by a normal vector and an orientation, i.e. we have vectors $\vec{n}_1, \ldots, \vec{n}_t \in \mathbb{R}^3$ and orientations $\vec{o}_1, \ldots, \vec{o}_t \in \mathbb{R}^3$. 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].

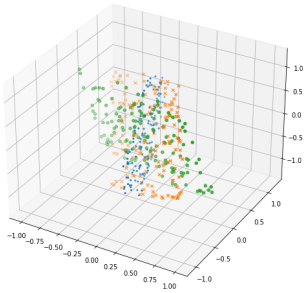
[7] Source: Jason Davies, https://www.jasondavies.com/maps/random-points/. Accessed on December 6th, 2020.



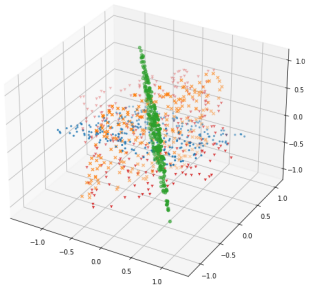
(a) Dataset 0, 3×30 points.



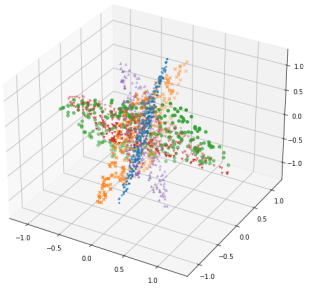
(b) Dataset 1, 3×80 points.



(c) Dataset 2, 3×150 points.



(d) Dataset 3, 4×250 points.



(e) Dataset 4, 5×300 points.

Figure 2: Generated datasets. Every set contains a number of planes with an equal amount of points. Noise level is $\epsilon = 0.01$.

The vectors are computed by

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

where $\text{norm}(\vec{h}) = \vec{h}/||\vec{h}||$ ($\vec{n}_i$ has already normalized length, since it describes a point on the unit-sphere). Secondly, 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.

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

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

$$\begin{aligned} c(\{u, v, w\}) &= 0, \\ c'(\{u, v, w\}) &= \text{d}(\{u, v, w\}, (0, 0, 0)) - \tau, \\ \text{d}(\{u, v, w\}, p) &= \frac{\vec{n} \cdot (u - p)}{||\vec{n}||}, \quad \vec{n} = (v - u) \times (w - u), \end{aligned}$$

where $\text{d}(\{u, v, w\}, p)$ is the distance between the plane that contains u, v and w and p ("$\cdot$" is the dot-product in this case) 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 set and if the distance is large, it is considered "bad".

run	dataset	**is**	N	M	**or** [s]	**iter**	**or**/**iter** [s]
0-S-S	0	1	40	1000	3.24	159	0.020
0-E-S	0	1	–	1000	24.66	103	0.239
0-S-E	0	1	40	–	5.54	191	0.029
0-E-E	0	1	–	–	18.30	68	0.269
1-S-S	1	1	40	5000	16.68	439	0.038
1-E-S	1	1	–	5000	949.42	375	2.532
1-S-E	1	1	40	–	59.18	461	0.128
1-E-E	1	1	–	–	473.91	143	3.314
2-S-S	2	1	50	5000	47.41	988	0.048
3-S-S	3	1	50	10000	321.57	4532	0.071
4-S-S	4	1	50	15000	1078.05	10000	0.108
2-S-SL	2	1	50	500	48.06	2341	0.021
2-SL-S	2	1	5	5000	65.49	8498	0.008
2-S-S-IP10	2	10	50	5000	71.68	1440	0.050
2-S-S-IP50	2	50	50	5000	77.64	1556	0.050
2-S-S-IP100	2	100	50	5000	101.04	2035	0.050

Table 1: Every run with the corresponding dataset it was executed on, the number of **i**nitial **s**ets in the partition, 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 were computed explicitly. For each run, τ was set to 0.1 and the stopping criterion was chosen in such a way that the randomized runs take at least 50 iterations and stop when the correct number of sets was reached. The deterministic runs (N and M both "–") search until no improving move exists.

The algorithms are evaluated in the four following settings on the datasets seen in figure 2 (all runs shown in table 1). 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 on the number of iterations. Finally, the randomized greedy search algorithm is initialized with random 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[8], the middle rows contain the used time per iteration – the

[8] If the reduced costs are negative in an iteration, then this means that none of the considered

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 sets in the partitions per iteration. All runs (except for the runs in setting 4) start with a partition where all vertices have the same set.
An observation that can be made beforehand with respect to all considered settings is that all instances of the greedy search with sampling tend to increase the number of sets in the partitions in the beginning while decreasing them in the end. This might be caused by the definition of the cost function: if many 3-ary subsets of vertices share the same set in the partition even if they are assigned high costs, it might be better to just split them up. Similarly, if the number of sets in the partitions is high, the probability that a vertex is assigned a set that "fits", i.e. yields negative costs, is higher.

Setting 1. Deterministic and randomized greedy search are compared on datasets 0 and 1, with 3×30 (runs 0-S-S, 0-E-S, 0-S-E, 0-E-E) and 3×80 (runs 1-S-S, 1-E-S, 1-S-E, 1-E-E) points respectively. 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 sets in the partitions (figure 3, 4 middle row). 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. However, the number of iterations does not increase significantly when decreasing the amount of neighbours (compare 0-S-S with 0-E-S or 1-S-S with 1-E-S), which suggests that the probability of selecting a random neighbour that lowers the costs is relatively high. This would mean that in every iteration, there are multiple ways of selecting a "good" neighbour, instead of only one. Computing the reduced costs for every possible pairing of vertices also takes a lot of time, as can be seen in run 1-S-S in comparison to 1-S-E, but also has no meaningful impact on the number of iterations.

Setting 2. We want to check the performance of the randomized search when considering comparatively larger problem instances, i.e. the partitioning of dataset 2, 3 and 4, see table 2. Here, the size of the considered neighbourhood for runs 2-S-S, 3-S-S and 4-S-S (figure 5) is set to $N = 50$ neighbours, but the number of samples to compute the reduced costs was increased with growing problem size ($5000, 10000$ and 150000 respectively). The performance of the randomized algorithm stays consistent with respect to every problem instance and yields an almost constant time requirement in each iteration. Only in the beginning there is a slight increase in the used time required to compute the (estimated) reduced costs (figure 5, middle row). This has to do with the definition of the cost-function: if there is only a low number of sets in the partitions, the likelihood of sampling a 3-ary subset that contains vertices which share the same set is high. Since only the computation of c' is expensive (in comparison to c), this yields an increased computational load.

neighbours yields an improvement over the current assignment.

Setting 3. The third setting compares the impact of small neighbourhoods (2-SL-S, $N = 5$) and small sample sizes (2-S-SL, $M = 500$) on the number of iterations/overall runtime with "ordinary" sizes (2-S-S, $N = 50, M = 5000$). See figure 6 for runs 2-SL-S and 2-S-SL and figure 5 for 2-S-S. Both for small sample sizes and small neighbourhoods, one can see that the number of iterations increases (figure 6, bottom row). This is expected, since large neighbourhoods and large sample sizes enable precise decision making in each iteration, and therefore lower the overall amount of iterations. However, since computation time per iteration decreases with lower neighbourhood/sample sizes, it may be possible that the overall runtime decreases as well. This is the case for run 2-S-SL, as can be seen in table 2, but not for run 2-SL-S (although both settings do not decrease/increase the overall runtime significantly). Another observation is that the number of sets in the partitions per iteration yields a different shape for low neighbourhood sizes (run 2-SL-S, bottom row): a stark increase in the beginning with a small gradual decrease afterwards.

Setting 4. The final setting examines the impact of different initializations on the number of iterations (runtime per iteration should be approximately equal, since N and M are left unchanged – therefore, the overall runtime should only depend on the number of iterations). Run 2-S-S-IP10 assigns every vertex one of 10 different sets, 2-S-S-IP50 assigns every vertex one of 50 different sets and 2-S-S-IP100 assigns every vertex one of 100 different sets (figure 7). As can be seen from figure 7, the number of iterations increases with growing number of initial assignments. A reason for this might be that the probability that two random vertices have the same set is low if the number of sets in the partitions is high. Since the costs were only defined on vertices sharing the same set, this might decrease the quality of the estimated costs.
Similarly to the previous setting, the number of sets in the partitions per iteration behave differently for the considered runs (figure 7, bottom row). For run 2-S-S-IP10, the curve (growth in the beginning, drop in the end) is similar to that of run 2-S-S. Run 2-S-S-IP50 increases the number of sets in the beginning only slightly, and run 2-S-S-IP100 stays almost constant before dropping to the optimal number.

From the above observations, we are able to draw the following conclusions with respect to the considered scenario. First, if M and N are chosen appropriately, the randomized version outperforms the deterministic version significantly when it comes to overall runtime. Especially the selection of a relatively low neighbourhood size yields a great time advantage, but the importance of selecting M not too high as well grows with more vertices (and planes). Second, the randomized greedy search yields consistent results if the problem size grows and allows for almost constant time requirements per iteration. Third, Selecting M relatively low yields the requirement for a greater number of iterations, but since every iteration is faster, the overall required time might decrease. Same for N. And finally, increasing the number of initial sets in the partition yields a greater number of iterations and a worse runtime.

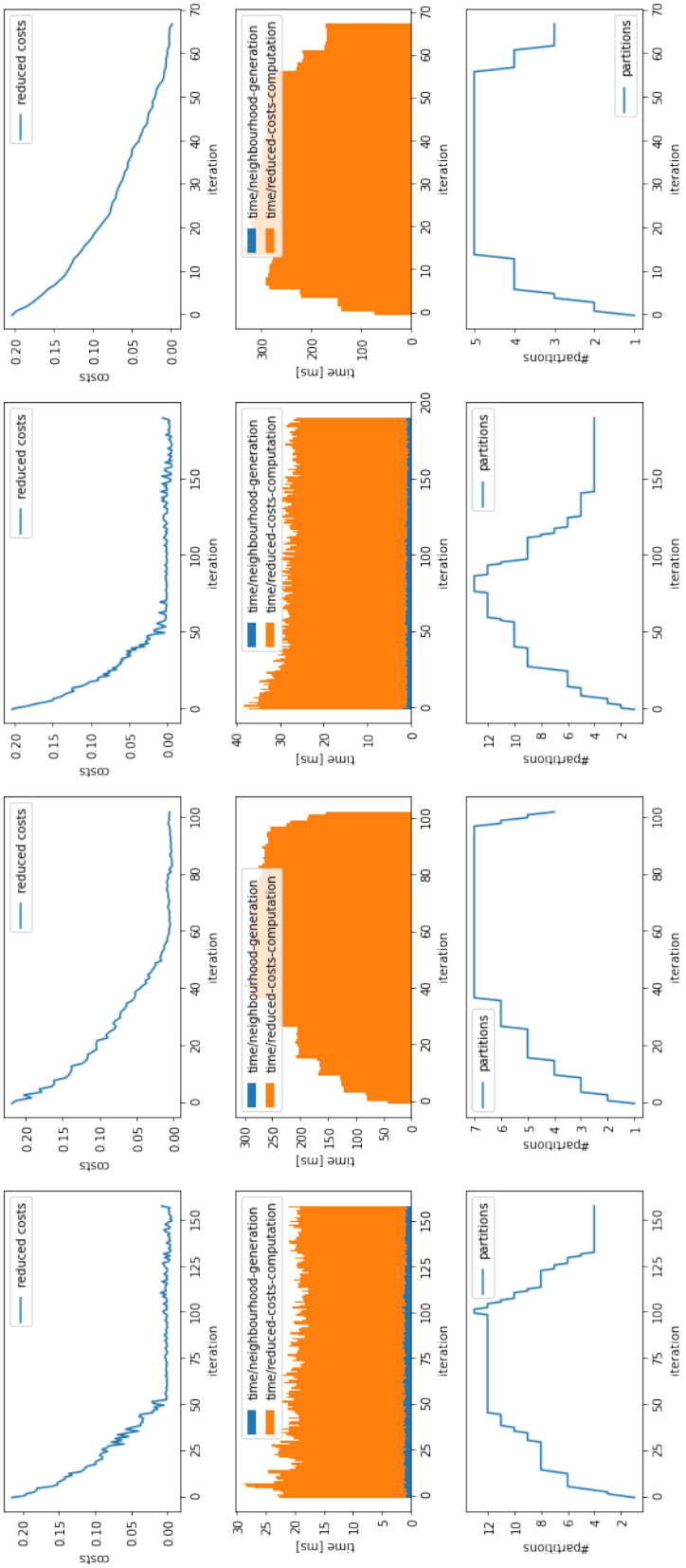


Figure 3: From left to right: run 0-S-S, 0-E-S, 0-S-E, 0-E-E.

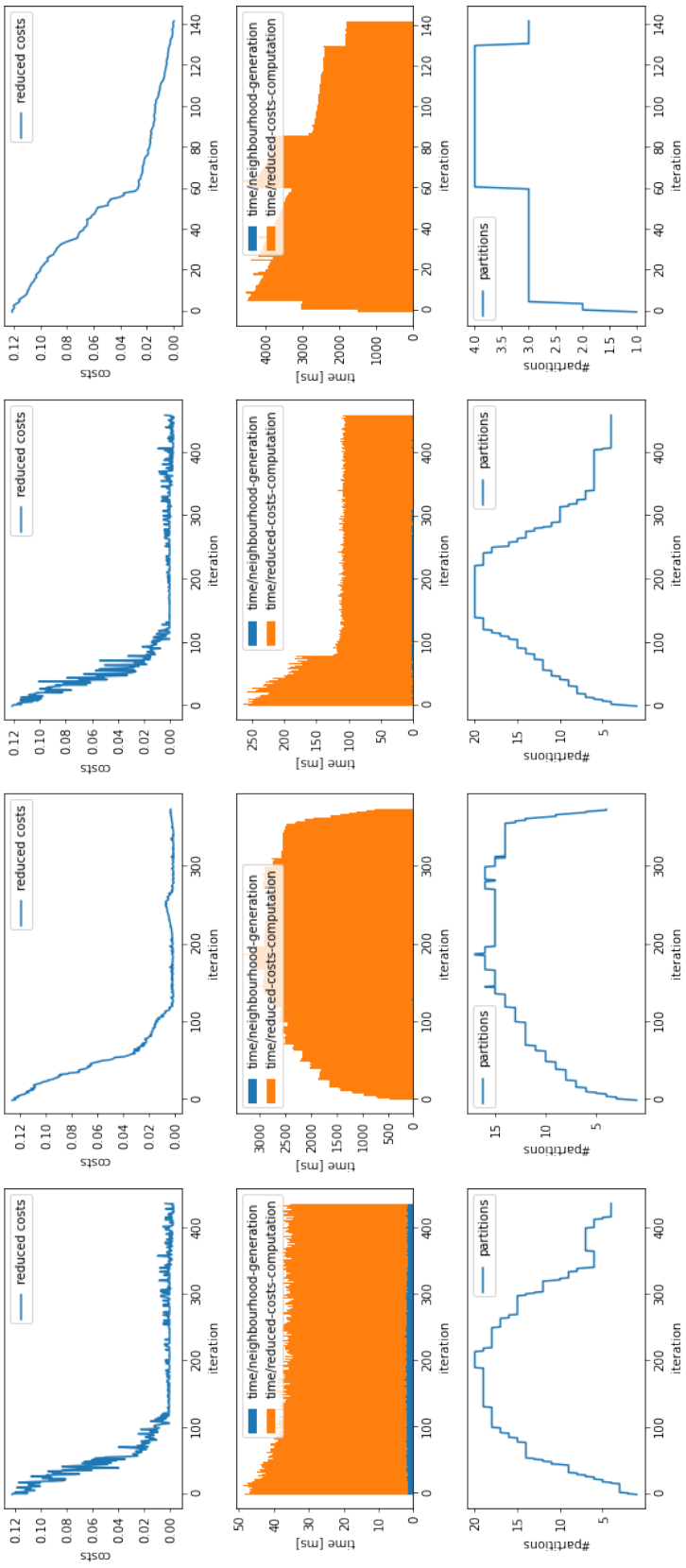


Figure 4: From left to right: run 1-S-S, 1-E-S, 1-S-E, 1-E-E.

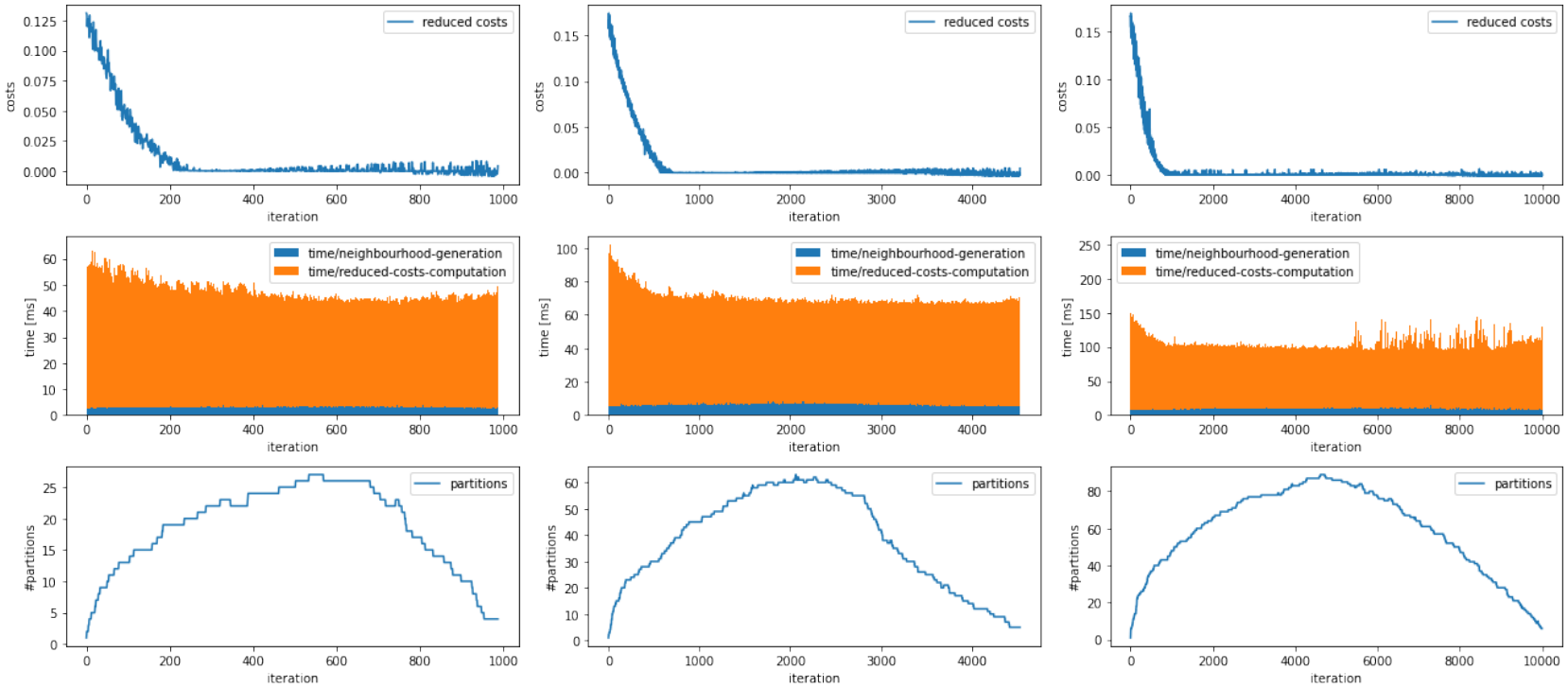


Figure 5: From left to right: run 2-S-S, 3-S-S, 4-S-S.

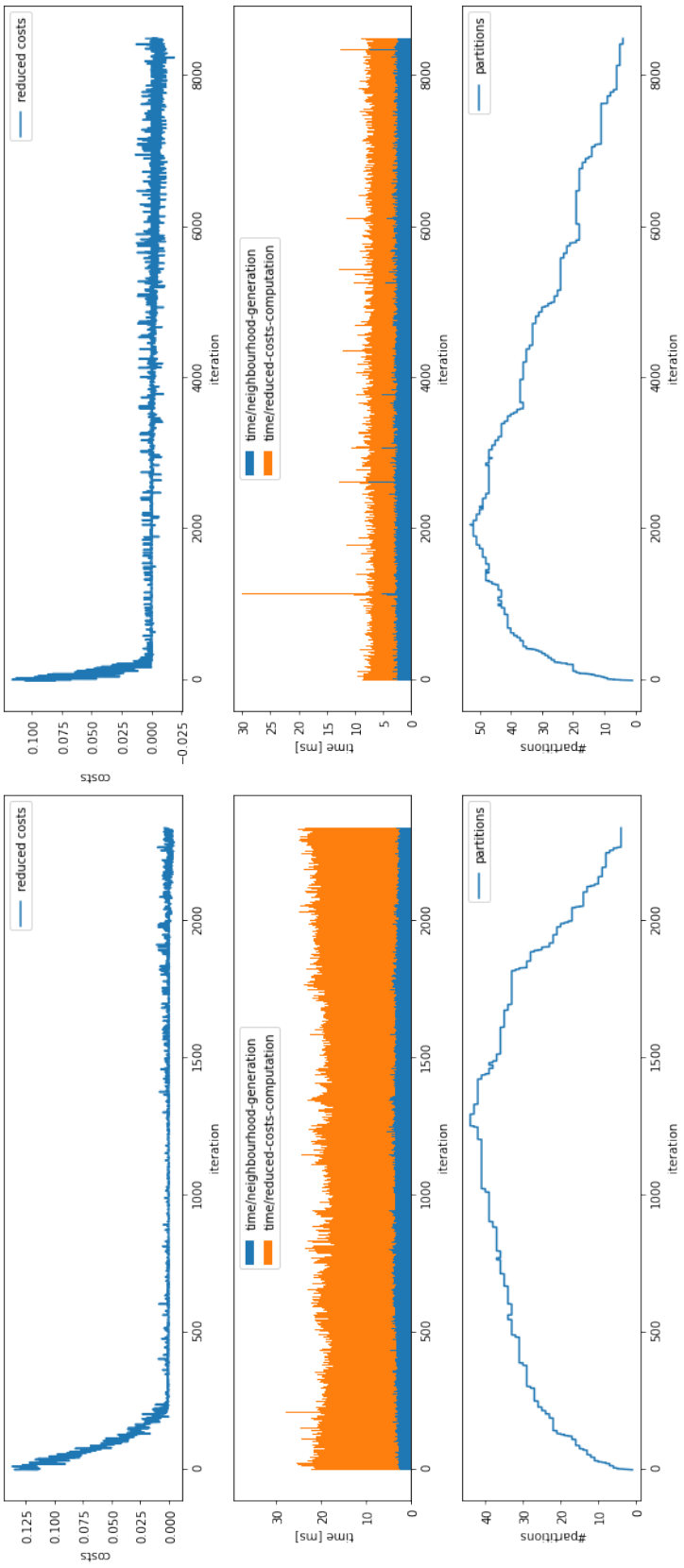


Figure 6: From left to right: run 2-S-SL, 2-SL-S.

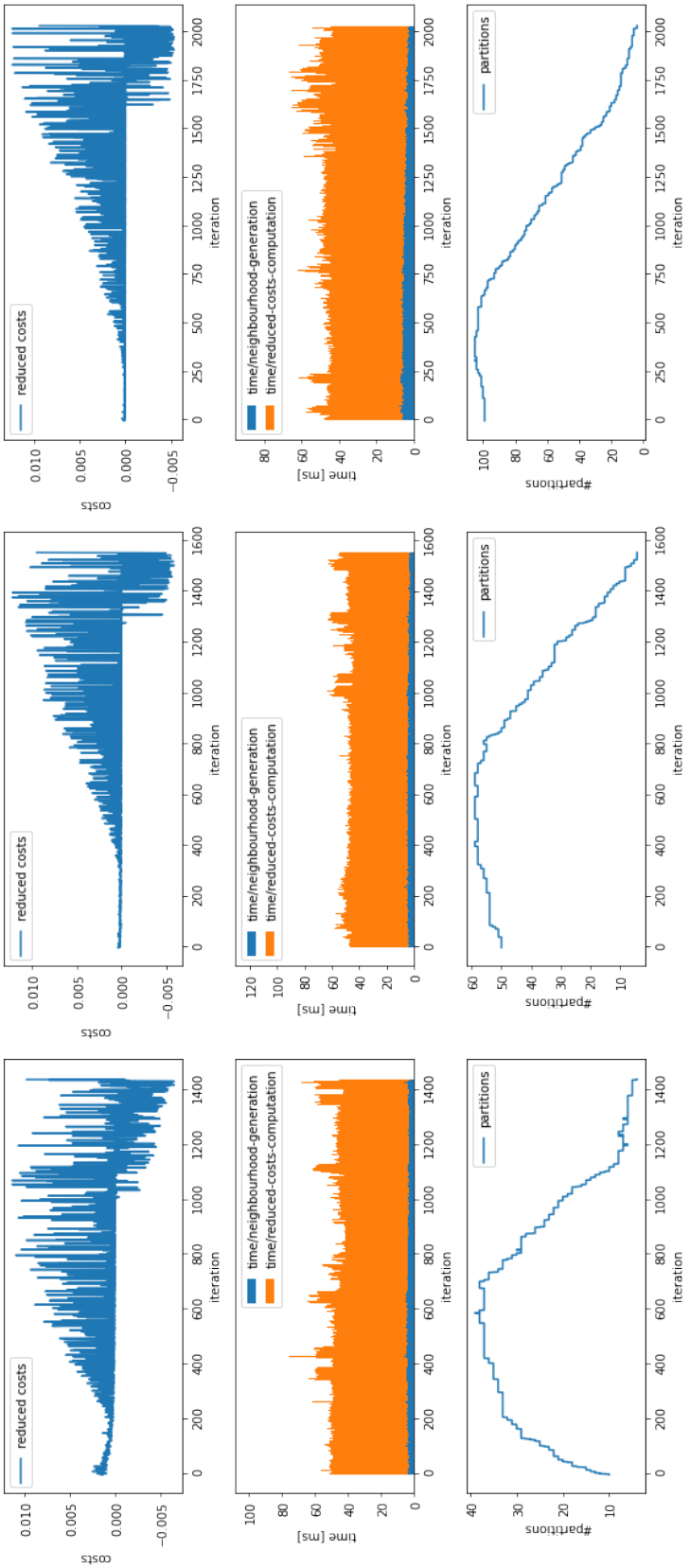


Figure 7: From left to right: run 2-S-S-IP10, 2-S-S-IP50, 2-S-S-IP100.

8 Conclusion and Possible Improvements

This project investigated a datastructure and (probabilistic) local-search methods for solving the partitioning-problem with respect to cost functions defined on 3-ary subsets of the given set. A special focus was put on the space- and time-efficiency of the used datastructure and fast improvements in each iteration. For this matter, the neighbourhood of a solution (partition) was defined over the set of possible "moves" of one vertex to another set, and an efficient way of enumerating the neighbourhood of a partition was given. In this regard, an algorithm for selecting random neighbours was defined, which allows for a comparatively fast way of sampling neighbours. In the end, the proposed datastructure and sampling-based methods were evaluated on a set of different experiments, which underlined the theoretical results.
Possible further investigations could include the following points. Using the theoretical results in Theorem 3.3, it is possible to check the equality of two partitions induced by indexings in time $O(|V|)$. This could be of use when implementing local search algorithms like Tabu-Search (Glover [1]) which keep track of previous solutions and require to compute whether new solutions are equal to past ones. On the same note, on could think of a Tabu-Search implementation that keeps track of which vertex was moved in the near past and then only considers neighbours that do not move these respective vertices. Another extension would be the implementation of simulated annealing, in which the fast way of selecting random neighbours might be helpful.

References

[1] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers operations research*, 13(5):533–549, 1986.

[2] 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.

A Proofs for Section 3 (Indexings as Proxies for Partitions)

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 $\Pi(\varphi)$. □

Theorem 3.3. *If φ, ψ are two indexings of V, then the following statements are equivalent:*

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

(b) there is a bijection $\lambda : \mathrm{image}(\varphi) \to \mathrm{image}(\psi)$[9] 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 $\psi(w) = \psi(v)$

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

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 $\lambda : \mathrm{image}(\varphi) \to \mathrm{image}(\psi)$ as follows. For all $k \in \mathrm{image}(\varphi)$, 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 $\mathrm{image}(\varphi)$ corresponds to a set in $\Pi(\varphi)$, and vice versa for $\mathrm{image}(\psi)$ and $\Pi(\psi)$, we get $|\mathrm{image}(\varphi)| = |\Pi(\varphi)| = |\Pi(\psi)| = |\mathrm{image}(\psi)|$. Since $\mathrm{image}(\varphi)$ and $\mathrm{image}(\psi)$ 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 $\varphi(v) = k \in \mathrm{image}(\varphi)$. 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 $\lambda(\varphi(w)) = \psi(w)$ 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 $\Pi(\varphi) = \Pi(\psi)$. □

B Proofs for Section 4 (Move-Operation on Indexings)

Lemma 4.1 (Pairwise Distinctiveness). *If $i \neq j$, then $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$.*

Proof. Note that every pair $(w, k) \in V \times \{1, \ldots, n\}$ 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 $i, j \in \{1, \ldots, m\}$ 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 $k_i = \varphi(v) \in \text{image}(\varphi) \backslash \{\varphi(w)\}$ (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, $k_j = \varphi(v) \in \text{image}(\varphi) \backslash \{\varphi(w)\}$. 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 3.3.
   (b) If $k_i = \varphi(v) \in \text{image}(\varphi) \backslash \{\varphi(w)\}$: 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 $\varphi_{w_j \to k_j}(w) = k_j \neq k_i = \varphi_{w_j \to k_j}(v)$. But that implies $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$ by Theorem 3.3.
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 $k_i = \varphi(v) \in \text{image}(\varphi) \backslash \{\varphi(w)\}$:
   (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 $[w_i]_\varphi = \{w_i, u, v, \ldots\}$: 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 3.3.
      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 3.3 and obtain $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$.
         B. $w_j = u$. Since $u \not< w_i$ 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 $k_j = \varphi(v) \in \text{image}(\varphi) \backslash \{\varphi(w_j)\}$ 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 3.3 yields $\Pi(\varphi_{w_i \to k_i}) \neq \Pi(\varphi_{w_j \to k_j})$.
   (b) If $k_i = \varphi(v) \in \text{image}(\varphi) \backslash \{\varphi(w_i)\}$, 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 3.3 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 3.3 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 3.3 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 $w_j \not< w_i$). 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 3.3 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 3.3 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). *If $w \in V$ and $1 \leq k \leq n$ and $\Pi(\varphi_{w \to k}) \neq \Pi(\varphi)$, then there is $1 \leq i \leq m$ with $\Pi(\varphi_{w \to k}) = \Pi(\varphi_{w_i \to k_i})$.*

Proof. Let φ be an indexing of V, let w be a vertex and $k \in \{1, \dots, n\}$. We want to show that if $\Pi(\varphi_{w \to k}) \neq \Pi(\varphi)$, then there is an $i \in \{1, \dots, m\}$ 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 $[v]_\varphi = \{v, s, \dots\}$. Line 6 in the algorithm directly enumerates $\varphi_{w \to k}$.
2. If $[w]_\varphi = \{w, u, \dots\}$. Since φ maps w and u to the same index, there is at least one index in $1, \dots, n$ that is assigned no vertex. But then $\mathcal{N} \neq \varnothing$. Again, there are the following options:
   (a) There is no $v \in V$ such that $k = \varphi(v)$:

i. If $[w]_\varphi = \{w, v, u, \dots\}$, 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 $k \in \mathrm{image}(\varphi)\backslash\{\varphi(w)\}$ 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)**.** *If* $1 \le i \le m$*, then* $\Pi(\varphi) \neq \Pi(\varphi_{w_i\to k_i})$.

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 $\varphi_{w_i\to k_i}$, $i \in \{1, \dots, m\}$. □