



Felix Windisch, BSc

A Novel Definition of Directed q -Nearness Comparative Analysis and Algorithms

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors

Robert Legenstein, Univ.-Prof. Dipl.-Ing. Dr.techn.

Institute of Theoretical Computer Science

Florian Unger, B.Sc. M.Sc.

Institute of Theoretical Computer Science

Graz, August 2024

Abstract

Network science is the craft of extracting information from relations with the help of various tools. This thesis focusses on one of the most long-standing of these tools: Q -analysis. With the recent introduction of directed Q -analysis, its use is no longer restricted to undirected graphs. But is this the only reasonable extension? And how do we compute it efficiently? In this work, we propose and explore an alternative definition of directed q -nearness. It aligns in the most prominent cases and exhibits subtle differences otherwise, but is in general equally strict.

Until now, the practical use of these definitions has been hindered by the lack of viable algorithms. We close this gap by presenting an efficient procedure to compute (q, i, j) -digraphs for both the original and novel definition. Under the assumption of constant maximal simplex dimension, these algorithms are shown to be asymptotically optimal.

The algorithms come paired with a high-performance Rust-based publicly available implementation, which outperforms previous implementations by six orders of magnitude. This enables, for the first time, the directed Q -analysis of large connectomes like the full statistical reconstruction of the Blue Brain Project and upcoming dense connectomic reconstructions.

Kurzfassung

Die Netzwerkforschung beschäftigt sich damit, Informationen aus Beziehungen zu gewinnen. Dabei werden unterschiedliche Werkzeuge eingesetzt, wie zum Beispiel Q -Analyse. Mit der neulichen Einführung von gerichteter Q -Analyse ist diese nun nicht mehr auf ungerichtete Graphen beschränkt. Die Frage besteht, ob es noch weitere mögliche Erweiterungen gibt. Mit dieser Arbeit führen wir eine alternative Definition für gerichtete Q -Analyse ein, die subtile Unterschiede aufweist.

Bisher war der praktische Einsatz dieser Techniken auf kleine Netzwerke beschränkt, da effiziente Algorithmen fehlten. Wir präsentieren effiziente Methoden, gerichtete q -Graphen für sowohl die neue als auch die originale Definition zu berechnen. Wenn man die maximale Simplex-Dimension als konstant annimmt, sind diese Algorithmen asymptotisch optimal in der Zeitkomplexität.

Zusätzlich stellen wir eine high-performance Implementierung in Rust bereit, welche die Berechnungen bis zu eine Million mal schneller durchführt als bisherige Implementierungen. Dadurch können zum ersten mal große Connectome wie die volle statistische Rekonstruktion des Blue Brain Project und zukünftige dichte biologische Rekonstruktionen gerichteter Q -Analyse unterzogen werden.

Acknowledgements

I would like to thank Professor Legenstein for supervising this thesis and giving me the freedom to explore topics at my own discretion. Special thanks go out to Florian Unger for giving the initial impulse on the topic and pushing me forward when things got difficult. This thesis would not have been possible without his tireless support. I am also thankful to Henri Riihimäki for laying the groundwork in this space and his enthusiastic support for the project. I would be remiss in not mentioning Jonathan Krebs, whose excellent implementation of the flagser algorithm formed the starting point for DirQ. Thanks should also go to my parents for supporting me in any way they can. Finally, I would like to thank my girlfriend Fiona. I could have written this thesis without her, but it would have been much less enjoyable.

Contents

1. Introduction	17
1.1. Classical Q -Analysis	17
1.2. Directed Q -Analysis	19
2. Preliminaries	25
2.1. Graphs and Simplices	25
2.2. Directed Q -Analysis	27
3. Conceptual Comparison of Definitions	31
3.1. Similarities	31
3.2. Differences	33
3.3. Combinatorial Analysis	37
3.3.1. Counting	37
3.3.2. Enumerating	38
3.4. Visualization	40
3.5. Limiting Simplex Dimension	44
4. Algorithms to Compute (q, i, j)-Digraphs	47
4.1. Naive Algorithm	48
4.2. Improved Algorithm to Compute \mathcal{Q}	50
4.2.1. Criterion [I]: Inclusion	50
4.2.2. Criterion [II]: $(q + 1)$ -simplices	53
4.2.3. Criterion [II]: Cofaces	55
4.2.4. Criterion [II]: Higher dimensional simplices	56
4.2.5. Summary	59
4.3. An Improved Algorithm for computing $\hat{\mathcal{Q}}$	59
4.3.1. Criterion [II]: $(\widehat{q, i, j})$ -near	59
4.3.2. Summary	62
5. Implementation	63
5.1. Representing Graphs and Simplices	63
5.2. Simplex IDs	63
5.3. Parallelization	64
5.4. Using DirQ	65
5.5. Benchmarks	66
6. Discussion and Outlook	69
6.1. Summary	69

6.2. Discussion	69
6.3. Outlook	71
Bibliography	73
A. Additional Algorithmic Ideas	75
A.1. Bottom-Up Algorithm	75
A.1.1. Computing Cofaces	75
A.1.2. Computing Supersimplices	76
A.1.3. Computing the Inclusion Graph	77
A.1.4. Full Algorithm	78
A.2. Matrix-based Algorithm	80
A.2.1. Example	83
A.2.2. Limitations and Drawbacks	85
A.2.3. Advantages	85

List of Figures

1.1.	Example to demonstrate the ideas of classic Q-analysis in the context of graph theory.	17
1.2.	The geometric representation of two 3-simplices glued to each other along a triangular side.	18
1.3.	Bipartite graph associating vertices from Figure 1.1a with the cliques that contain them.	19
1.4.	A directed 2-, 3-, 4-simplex and a flattened 4-simplex.	20
1.5.	Directed simplices interact in different ways.	20
1.6.	Possible 1-nearness configurations of two 2-simplices with implied directionality.	21
1.7.	Important structures of graphs become apparent under (q, i, j) -nearness.	21
2.1.	Directed graph and corresponding flag complex with selected face (∂_i) , coface (coF_i) and inclusion (\hookrightarrow) relations marked.	27
2.2.	Both definitions of directed q-nearness and the classic undirected definition.	28
2.3.	Chains or cycles of 1-near (2)-simplices for different values of i and j	29
2.4.	Example graph and associated $(1, 0, 2)$ -near-digraph.	29
3.1.	Modified version of definition in Figure 2.2a making use of the reflexivity of inclusion shows that both definitions are equivalent for $(q + 1)$ -simplices.	31
3.3.	Subset of flag complex that represents a counterexample to Lemma 3.2.2 for $(\widehat{q, i, j})$ -nearness.	35
3.4.	Heatmap showing the proportion of simplex pairs that share a (4)- / (5)-face and are $(4, i, j)$ -near.	40
3.5.	Two directed simplices sharing a face and the corresponding bipartite graph drawing.	41
3.6.	Arrow diagrams of simplex pairs that are $(\widehat{2, 1, 1})$ -near (left), $(2, 1, 1)$ -near (right) or both (middle). The structure of the $(2, 1, 1)$ direction for both definitions is highlighted in white.	42
3.7.	Arrow diagrams for structures that are contained in (q, i, j) -near simplex pairs with shared (q) -face.	44
4.1.	The (q, i, j) -digraph can be computed by finding all instances of patterns from Figure 2.2a in the flag complex.	47
4.2.	Schematic of intersection tests in Algorithm 2 (left) and 3 (right).	49
4.3.	Inclusion graph with results of Example 4.2.3 highlighted in red.	51

4.4.	(Left) The original definition of (q, i, j) -near. (Right) The equivalent alternative definition using the cofaces of (q) -simplices.	53
4.5.	Criterion [II] for higher dimensional simplices.	56
4.6.	(Left) The original definition of $(\widehat{q, i, j})$ -near. (Right) The equivalent alternative definition using cofaces.	60
5.1.	Graphical User Interface of DirQ	66
A.1.	Example of a directed acyclic graph induced by cofaces.	76
A.2.	Arrow diagrams of simplex pairs that are both $(\widehat{3, 0, \infty})$ -near.	86

List of Tables

3.1.	Heatmaps showing the percentage of (q, i, j) -near pairs of (7) -simplices (out of all possible pairs that share a (q) - or $(q + 1)$ -face) that are also $\widehat{(q, i, j)}$ -near for different values of q, i, j and $\dim(\alpha)$	39
5.1.	Benchmark results for the Rust implementation of the naive and efficient algorithm.	67
5.2.	Benchmarks for different parallelization methods on the Blue Brain Project graph for different number of cores.	67

List of Algorithms

1.	Naive algorithm.	48
2.	Check $(\widehat{q, i, j})$ -nearness of two simplices.	48
3.	Check (q, i, j) -nearness of two simplices.	49
4.	Method for computing the inclusion graph.	52
5.	Compute E_{q+1} using coface maps.	54
6.	Construct the coface data structure.	55
7.	Compute (q, i, j) -nearness using coface maps and inclusion.	58
8.	The full improved algorithm for computing \mathcal{Q}	59
9.	Compute $(\widehat{q, i, j})$ -near relations using the inclusion graph	61
10.	Efficient algorithm for computing $\hat{\mathcal{Q}}$	62
11.	Routine to compute i -cofaces of one simplex	75
12.	Routine to compute $\delta_{\mathcal{Q}_{\hookrightarrow}}^+(\sigma)$ recursively.	77
13.	Routine to compute a subgraph of the inclusion graph for all simplices that contain σ	78
14.	Bottom-Up algorithm to compute \mathcal{Q}	78
15.	Bottom-Up algorithm to compute $\hat{\mathcal{Q}}$	79

1. Introduction

This section aims to introduce the concept of Q-Analysis, disambiguate it from other network science techniques and motivate the directed extension.

1.1. Classical Q -Analysis

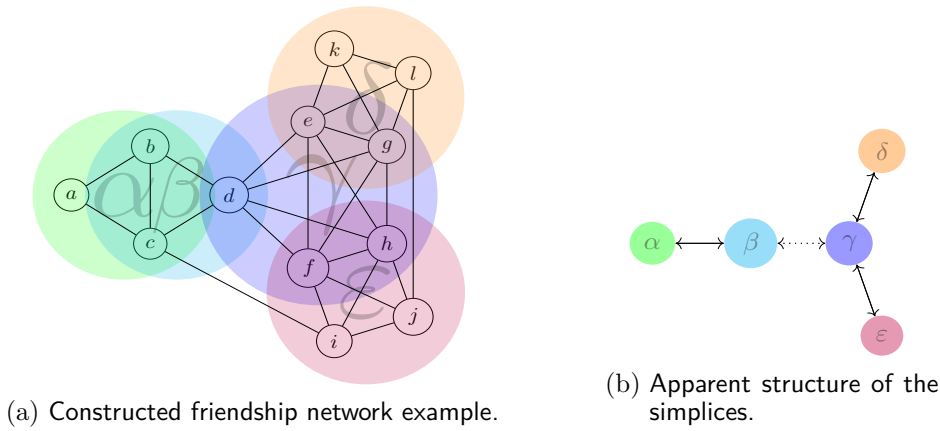


Figure 1.1.: Example to demonstrate the ideas of classic Q-analysis in the context of graph theory.

Q-analysis is a technique developed in the early 1970s by Ronald Atkin who often referred to it as “the algebra of relations” [Atk74]. In that spirit, we will present his ideas using the constructed example of a friendship network G shown in Figure 1.1a. Each vertex represents an individual and the edges denote mutual friendship. In the above graph, five tight-knit friend groups (denoted by greek letters) may be found.

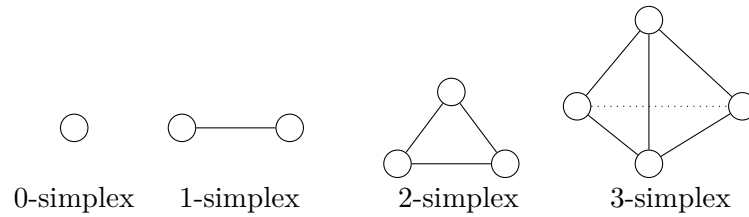
These densely connected groups could be identified by classic network science techniques such as graph clustering and community detection. In contrast, Q-analysis takes a narrower view and considers only cliques (meaning fully connected as opposed to densely connected subgraphs), such as the five groups of friends depicted in G . The focus is then placed on how these cliques are connected.

Intuitively, it appears that clique γ is strongly connected to cliques δ and ε . The clique β shares a strong connection with α and a weaker connection with γ . This higher-level view is represented in Figure 1.1b. Indeed, cliques that share more nodes apparently have a stronger connection. In the sociological context, it seems logical that groups which have more members in common would be more tightly linked, motivating the

following definition on graphs: Two cliques are called q -near if they share $q + 1$ vertices between them. It follows that β is 0-near to γ , while γ is 1-near to both δ and ε . But why does q -nearness require $q + 1$ shared simplices? The reason originates in geometry.

Geometrical Perspective

From a geometric point of view, the vertices of a clique with n vertices can be viewed as points in some $(n - 1)$ -dimensional space. The convex hull of those points would then contain edges between any two points, mirroring the edges of the clique. This convex hull is also referred to as a simplex. For a clique of 3 vertices e.g., the 3 points in 2D space would form a 2-simplex or triangle. The following shows examples of simplices in various dimensions:



For this reason, cliques in a graph are usually referred to as *simplices* in algebraic topology. As such, two simplices would be considered q -near iff they share a q -dimensional geometric face. One may consider these two simplices to be "glued" along the (q) -face. For example, two 3-simplices that are 2-near form two tetrahedrons that are "glued" together on one of the triangular sides (see Figure 1.2).

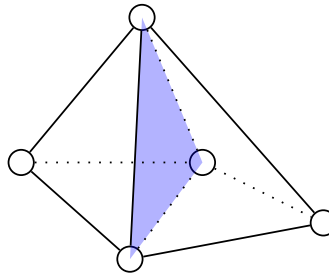


Figure 1.2.: The geometric representation of two 3-simplices glued to each other along a triangular side.

Remark 1.1.1. Another view of the friendship graph might associate each individual with the cliques they are a part of, as shown in Figure 1.3 in the form of a bipartite graph. Note that this relation captures all information necessary to find the q -structure of G . It can be seen as a kind of intermediary step between the original graph and Figure 1.1b. As such, the discussed techniques may be used directly on any binary relation between two finite sets (in this case, vertices and cliques) that need not necessarily be derived from a graph. In fact, that is how Q-analysis was first introduced. This work on the other hand only focuses on the relations of simplices within graphs.

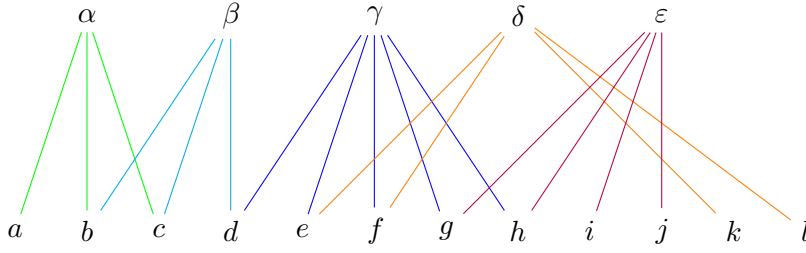


Figure 1.3.: Bipartite graph associating vertices from Figure 1.1a with the cliques that contain them.

1.2. Directed Q -Analysis

As we have seen, undirected q -analysis enables structural insights on undirected graphs, but many real-world systems can only be modelled using one-way relationships. Consider for example a directed graph of websites with edges where one page links to another. The classic version of Q -analysis may be applied to directed graphs by simply ignoring the direction of the edges. This leads to considerable loss of information, which is why new techniques were needed. Acknowledging this, Riihimäki recently ([Rii23]) proposed a novel approach to Q -analysis on directed graphs a few decades after Atkin's original work.

Directed Simplices

The simplices of a directed graph are no longer classical cliques, but instead are thought of as redundant flows in the graph. Each directed simplex is then formalized as an ordered subset of vertices such that there is an edge from any vertex lower in the order to any vertex higher in the order. The first vertex is called the source s and the last vertex the sink t of the simplex. Directed n -simplices have the following properties:

- (*Density*) The underlying undirected graph of a directed simplex is a clique.
- (*Flow*) Any walk on the edges of the simplex starting at s will reach t in at most n steps.
- (*Redundancy*) In any directed simplex of dimension n , a path from s to t will exist even after any $n - 1$ edges are removed.

Figure 1.4 shows examples of directed simplices.

In classical Q -analysis, the vertices of a simplex are interchangeable. It only matters how many, not which vertices are shared between cliques. With directed simplices however, it can make a great difference whether e.g. the source vertex or the sink vertex is shared.

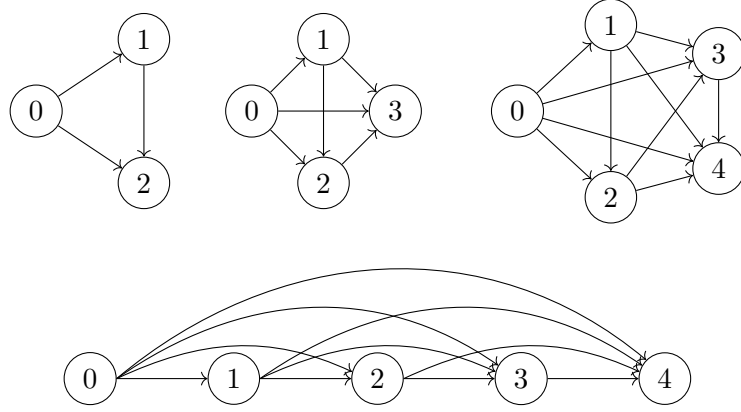


Figure 1.4.: A directed 2-, 3-, 4-simplex and a flattened 4-simplex.

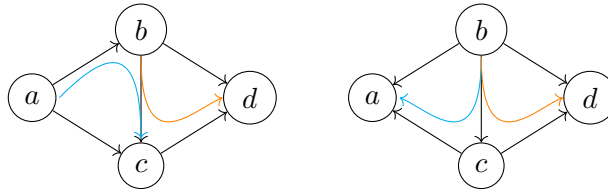


Figure 1.5.: Directed simplices interact in different ways.

Example 1.2.1. Consider the two graphs in Figure 1.5. On the left, simplices (abc) and (bcd) share the edge (bc) and the "flow" leads from the source a to the sink d . On the right, the edge is shared by (bca) (note the order) and (bcd) . The simplex "flow" starts at b and splits towards the two sinks a and d .

A new definition of directed q -nearness is needed to reflect this additional complexity. For any two directed $(q + 1)$ -simplices that share a (q) -face, one vertex of each is not shared. The indices of those vertices in the order of their simplices can be denoted as i and j respectively. Any combination of i and j leads to a different configuration of edge directions. Figure 1.6 depicts (2) -simplices that share an edge for every pair of i and j , along with arrows that approximately represent the flow of the simplices. Clearly, all of these configurations express different interactions between the directions of the simplices. In order to differentiate these cases, the notion of q -nearness is expanded to (q, i, j) -nearness. We observe the definition for the special case of simplices with exactly $q + 2$ vertices.

Directed (q, i, j) -nearness

Two directed $(q + 1)$ -simplices σ, τ are (q, i, j) -near iff σ without the i th vertex is equal to τ without the j th vertex or they are equal: $\sigma \setminus \sigma_i = \tau \setminus \tau_j$ or $\sigma = \tau$.

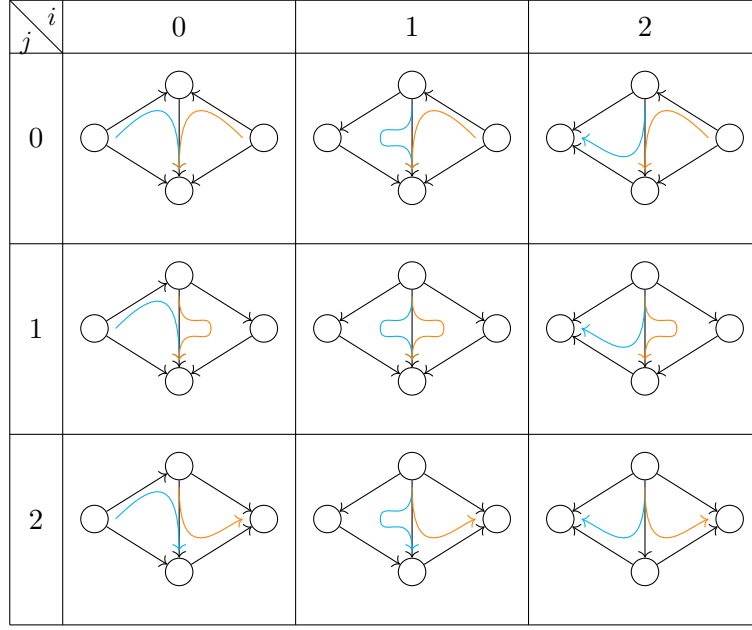
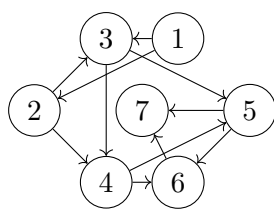


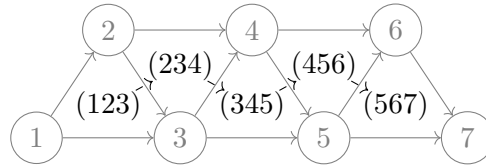
Figure 1.6.: Possible 1-nearness configurations of two 2-simplices with implied directionality.

Intuition

When thinking of directed simplices as flows or arrows, (q, i, j) -nearness intuitively becomes the relation of arrows being glued together at different points. For example, two simplices that are $(q, 0, 0)$ -near start with separated sources and then combine together, whereas $(q, q + 1, q + 1)$ -near simplices start at a common source and then split apart. Another important case is $(q, 0, q + 1)$ -nearness, where the sink of one simplex transitions into the source of another, basically extending the flow. Examples for these cases can be found in the corners of Figure 1.6.



(a) Graph example



(b) The same graph in different embedding in gray along with maximal simplices connected by arrows representing $(1, 0, 2)$ -nearness.

Figure 1.7.: Important structures of graphs become apparent under (q, i, j) -nearness.

Example 1.2.2. Directed (q, i, j) -nearness can help identify particular structures in graphs. Figure 1.7 shows a graph that represents a longer redundant flow. This direct-

edness can be hard to spot in the wrong embedding. When we consider the $(1, 0, 2)$ -nearness relations of the maximal simplices, we find that they form a chain, indicating a strong flow in a particular direction. For example, (123) is $(1, 0, 2)$ -near to (234) , because (123) without the first (index 0) vertex and (234) without the last (index 2) vertex form the same 1-simplex (or edge) (23) .

Higher dimensional Simplices

This definition of (q, i, j) -nearness is natural to apply to $(q + 1)$ -dimensional simplices, as it covers all possible permutations. But it is not sufficient to decide whether a $(q + n)$ -simplex is (q, i, j) -near to some other $(q + m)$ -simplex. Adding $n + m$ indices to the definition, one for each vertex that is not shared, is impractical. The definition would become too narrow and difficult to interpret. Dropping i and j from the definition would just revert to classical undirected q -nearness. We consider two approaches to extend (q, i, j) -nearness to higher dimensions.

- In [Rii23] the definition is straight-forwardly extended as follows: Two simplices σ, τ are (q, i, j) -near if they share $q + 1$ vertices, none of which are σ_i or τ_j , and the remaining agree on the order.
- In this work, a novel extension is presented: Two simplices σ, τ are considered (q, i, j) -near if σ contains a $(q + 1)$ -simplex that is (q, i, j) -near to any $(q + 1)$ -simplex contained in τ according to the previous definition for $(q + 1)$ -simplices.

Both definitions follow the previously discussed rules for $(q + 1)$ -simplices.

These two definitions and their differences are central to this work.

Algorithmic Aspects

Determining whether two particular simplices are (q, i, j) -near can be accomplished by a simple procedure (formalized in Algorithm 3). This procedure can be used to iteratively find all (q, i, j) -nearness relations in the graph by checking every possible pair of simplices. This is infeasible, as the number of simplices in a graph scales with the factorial of the number of vertices. However, the combinatorial structure of the simplices may be exploited to reuse computations and greatly increase efficiency. In fact, we develop an asymptotically optimal algorithm to enumerate all (q, i, j) -nearness relations with time complexity linear in output size.

Outline

The aim of this thesis is twofold:

- We first present a novel definition of directed q -nearness and compare it with the established notion. Along the way, we develop intuitive visualizations to think about directed q -nearness and prove various properties.

- We design time optimal algorithms for both definitions and provide efficient open-source implementations in Rust ¹. By providing a speedup of more than six orders of magnitude, we enable rapid analysis of networks relevant to Computational Neuroscience.

Section 2 introduces the previous notions formally in exhaustive detail. In section 3, differences and similarities of both definitions are discussed. Additionally, methods for visualization are presented. Several algorithms to enumerate all (q, i, j) -nearness relations for both definitions are detailed and compared in section 4. Section 5 dives into aspects of implementing and optimizing the algorithms on real machines and compares the various implementations on benchmarks. Additional algorithmic notions that may be useful for further exploration are contained in Appendix A. A discussion of directed q -nearness and its potential in section 6.1 completes the work.

¹<https://github.com/FelixWindisch/DirQ>

2. Preliminaries

The following definitions for graphs and simplices are in alignment with [UKM23].

2.1. Graphs and Simplices

Definition 2.1.1 (Directed and Undirected Simple Graphs).

- A *simple (undirected) graph* is a pair $G = (V, E)$ of a finite set of vertices V and a symmetric relation $E \subseteq (V \times V) \setminus \Delta_V$, where $\Delta_V := \{\{v, v\} \mid v \in V\}$.
- A *simple directed graph* is a pair $G = (V, E)$ of a finite set of vertices V and a relation $E \subseteq (V \times V) \setminus \Delta_V$, where $\Delta_V = \{(v, v) \mid v \in V\}$.
- A *directed, oriented graph* is a pair $G = (V, E)$ of a finite set of vertices V and a relation $E \subseteq (V \times V)$, where $\forall (s, t) \in E : (t, s) \notin E$.
- The set of *successors* of a vertex v in a simple directed graph $G = (V, E)$ is denoted as $\delta^+(v) := \{w \in V \mid (v, w) \in E\}$. Analogously, the *predecessors* are defined as $\delta^-(v) := \{w \in V \mid (w, v) \in E\}$.
- Let $G = (V, E)$ and $S = (V_S, E_S)$ be (directed or undirected) graphs with $V_S \subseteq V$ and $E_S \subseteq E$. Then S is called a subgraph of G .

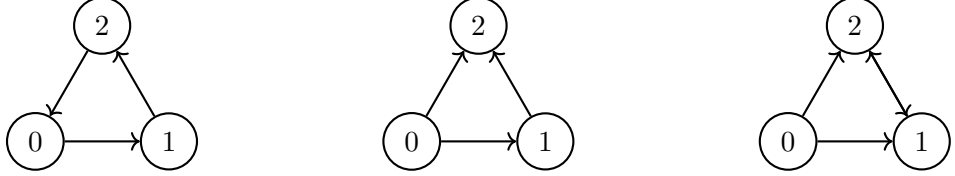
By excluding Δ_V from E , we exclude self-loops from v to itself. Since we will exclusively use simple directed graphs, we will often refer to them simply as *graphs*.

Definition 2.1.2 (Cliques and Simplices). Let $G = (V, E)$ be a directed graph.

- A *d-simplex* is a totally ordered subset $(v_0 v_1 \dots v_d)$ of V such that additionally $(v_i, v_j) \in E \forall i < j$. To distinguish simplices from other sets, they will be denoted as $(v_0 v_1 \dots v_d)$.
- The *dimension* of a simplex σ , denoted $\dim(\sigma)$ is equal to $|\sigma| - 1$.
- We refer to two simplices σ, τ as *parallel* if for all pairs $(v_i, v_j) \in V_\sigma \cap V_\tau$, either $v_i < v_j$ or $v_i > v_j$ in the total order of both σ and τ . In other words, their directions do not contradict.
- A simplex τ is a *face* of σ , denoted $\tau \hookrightarrow \sigma$, if all vertices of τ are contained in σ and they are parallel. This relation is reflexive, as every simplex includes itself. We also say σ *includes* τ or that σ is a *parent* or *supersimplex* of τ . A simplex α is a *shared face* of σ and τ iff $\sigma \hookleftarrow \alpha \hookrightarrow \tau$.

- The simplex $\sigma \setminus \tau$ refers to the maximal face of σ that does not contain any vertices of τ and inherits the order from σ .
- The intersection of two parallel simplices $\sigma \cap \tau$ is a simplex over the set of all vertices contained in both σ and τ with inherited order.

Example 2.1.3. A d -simplex is a subgraph of a $(d+1)$ -clique. But a $(d+1)$ -clique does not have to contain a d -simplex: Indeed, there can be zero, one or even several d -simplices per $(d+1)$ -clique:



Here we see three 3-cliques: The one on the left contains no 2-simplex, as it is cyclic. The one in the middle contains exactly (012) . The one on the right even exhibits two 2-simplices $((012)$ and $(021))$.

Definition 2.1.4 (Directed Flag Complexes and Face Maps).

- Let $G = (V, E)$ be a simple directed graph and D the dimension of the highest-dimensional simplex in G . The directed flag complex Σ is a tuple of length $D+1$ storing all the d -dimensional simplices in its respective fields, i.e. $\Sigma := (\Sigma_0, \Sigma_1, \dots, \Sigma_D)$. Note that $\Sigma_0 = V$ and $\Sigma_1 = E$. The notation $\Sigma_{>q}$ is often used to denote the set of simplices with dimension greater than q .
- Let $\Sigma_n \ni \sigma = (\sigma_0 \dots \sigma_n)$. The *face map*

$$\hat{\partial}_i(\sigma) := \begin{cases} \sigma \setminus (\sigma_i) & \text{if } i \leq n \\ \sigma \setminus (\sigma_n) & \text{if } i > n \end{cases}$$

associates the simplex with its i th face. Whenever $i \leq n$ can be guaranteed, we omit the hat and use $\partial_i(\sigma) := \sigma \setminus (\sigma_i)$.

- Let $\sigma \in \Sigma_p$. The **coface** operator $\text{coF}_i(\sigma) = \{\tau \in \Sigma_{p+1} \mid \partial_i(\tau) = \sigma\}$ is the left inverse of ∂_i . Note that coF_i yields a set of simplices and thus is only a one-sided inverse of ∂_i .
- Let $\text{faces}(\sigma, n) := \{\tau \in \Sigma_n \mid \tau \subset \sigma\}$ denote the set of all n -dimensional faces of σ .

The problem of computing directed flag complexes from graphs has been thoroughly explored in [Lü+20] and [Rei+17]. As such, we will not describe the process and will consider the flag complex as given. In the actual implementation, we use techniques borrowed from [Lü+20]. Examples for different relations on a flag complex are demonstrated in Figure 2.1.

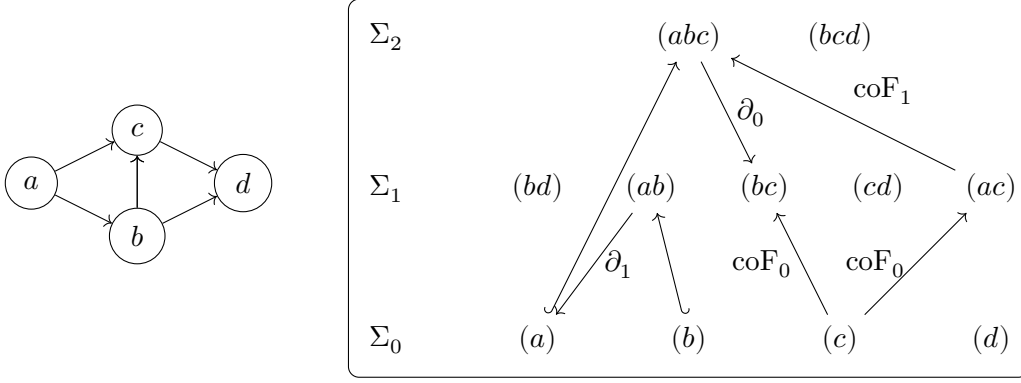


Figure 2.1.: Directed graph and corresponding flag complex with selected face (∂_i) , coface (coF_i) and inclusion (\hookrightarrow) relations marked.

2.2. Directed Q -Analysis

Riihimäki [Rii23] introduced this expanded definition for q -nearness of simplices in directed graphs:

Definition 2.2.1 (original directed q -nearness). Let Σ be a directed flag complex and (σ, τ) be an ordered pair of simplices $\sigma, \tau \in \Sigma_{>q}$. Let $(\hat{\partial}_i, \hat{\partial}_j)$ be an ordered pair of face maps. Then σ is $(\widehat{q, i, j})$ -near to τ if either of the following conditions is true:

[I] $\sigma \hookrightarrow \tau$,

[II] $\hat{\partial}_i(\sigma) \hookleftarrow \alpha \hookrightarrow \hat{\partial}_j(\tau)$ for some $\alpha \in \Sigma_q$.

Remark 2.2.2. Note that the shared face α is also an ordered set of vertices. As such, the simplices (abc) and (cbd) (in this case, b and c have a bidirectional edge) are not $(\widehat{1, 0, 2})$ -near, because $(bc) \neq (cb)$.

We introduce a modified version of this definition:

Definition 2.2.3 (new directed q -nearness). Let Σ be a directed flag complex and (σ, τ) be an ordered pair of simplices $\sigma, \tau \in \Sigma_{>q}$. Let (∂_i, ∂_j) be an ordered pair of face maps with $i, j \in \{0, \dots, q+1\}$. Then σ is (q, i, j) -near to τ if either of the following conditions is true:

[I] $\sigma \hookrightarrow \tau$,

[II] There exist a q -simplex $\sigma \in \Sigma_q$ and two $(q+1)$ -simplices $\mu_\sigma \hookrightarrow \sigma, \mu_\tau \hookrightarrow \tau$ such that $\partial_i(\mu_\sigma) = \alpha = \partial_j(\mu_\tau)$.

Figure 2.2 gives a schematic overview of criterion [II] for both the new and original definition on directed graphs, as well as classic undirected q -nearness.

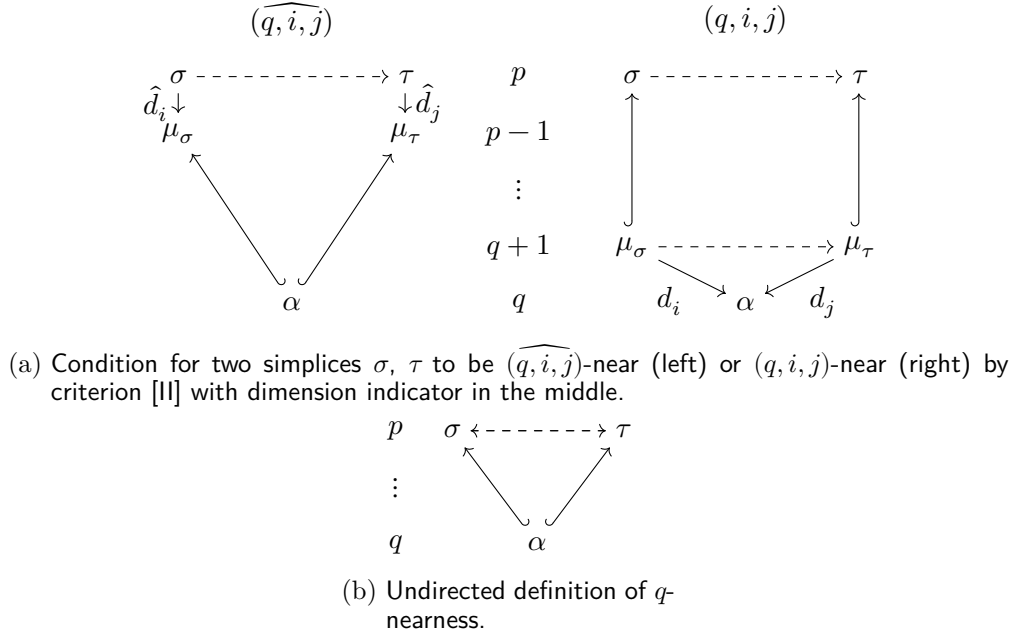


Figure 2.2.: Both definitions of directed q -nearness and the classic undirected definition.

Henceforth, we may refer to simplices as being (q, i, j) -near by criterion [I] (inclusion) or criterion [II] (shared face) for both definitions. Hooked arrows ($\sigma \hookrightarrow \tau$) denote q -nearness by criterion [I], while dashed arrows ($\sigma \dashrightarrow \tau$) denote q -nearness by criterion [II].

Remark 2.2.4. The old definition makes use of boundary maps $\hat{\partial}_i(\sigma)$, because i might be greater than $|\sigma|$. In the new definition, face maps are only applied to $(q+1)$ -simplices, so we can restrict $i \in \{0, \dots, q+1\}$ and use ∂_i . In fact, $(q, 0, q+2)$ -near is equivalent to $(q, 0, q+1)$ -near, but $(q, 0, \widehat{q+2})$ -near is not always equivalent to $(q, 0, \widehat{q+1})$ -near. Oftentimes $\hat{\partial}_\infty$ is used to denote the face map that removes the last vertex.

Definition 2.2.5 ((q, i, j) -Digraphs). Let G be a simple, directed graph with directed flag complex Σ . The (q, i, j) -digraph $\mathcal{Q} = \{\Sigma, E^{\mathcal{Q}}\}$ of G contains an edge (σ, τ) if $\sigma \in \Sigma$ is (q, i, j) -near to $\tau \in \Sigma$. Analogously, $\hat{\mathcal{Q}} = \{\Sigma, E^{\hat{\mathcal{Q}}}\}$ contains connections between $(\widehat{q, i, j})$ -near simplices.

When referring to particular q -digraphs, they are denoted as the (q, i, j) - or $(\widehat{q, i, j})$ -digraph of G . For the sake of simplicity, we will omit self-loops in \mathcal{Q} and $\hat{\mathcal{Q}}$. Examples of (q, i, j) -near chains can be found in Figure 2.3 and a $(1, 0, 2)$ -digraph is depicted in Figure 2.4.

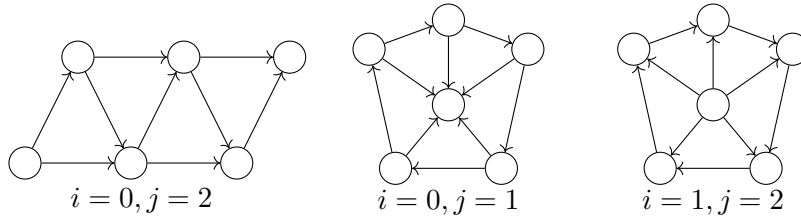


Figure 2.3.: Chains or cycles of 1-near (2)-simplices for different values of i and j .

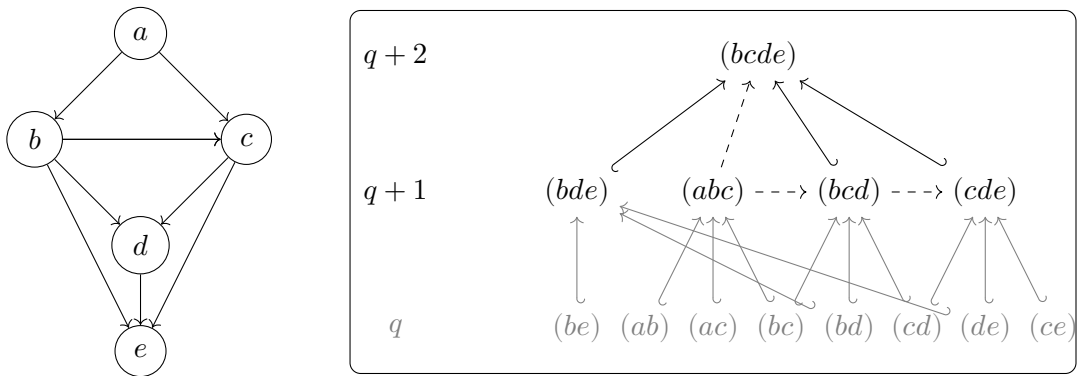


Figure 2.4.: Example graph and associated $(1, 0, 2)$ -near-digraph.

3. Conceptual Comparison of Definitions

In the following, we explore differences and similarities between the original definition $(\widehat{q, i, j})$ for q -nearness on directed graphs and our novel definition (q, i, j) .

3.1. Similarities

As mentioned in the introduction, both definitions are identical for $(q + 1)$ -simplices. The following lemma formalizes this statement.

Lemma 3.1.1. *If $\sigma, \tau \in \Sigma_{q+1}$ are (q, i, j) -near, they are also $(\widehat{q, i, j})$ -near and vice versa.*

Proof. The simplices $\sigma, \tau \in \Sigma_{q+1}$ are (q, i, j) -near exactly when $\partial_i(\sigma) = \partial_j(\tau) = \alpha$, because $\sigma \hookrightarrow \sigma$ and $\tau \hookrightarrow \tau$. This also means that $\partial_i(\sigma) \hookleftarrow \alpha \hookrightarrow \partial_j(\tau)$, which is sufficient to show σ is $(\widehat{q, i, j})$ -near to τ .

In turn, $\partial_i(\sigma) \hookleftarrow \alpha \hookrightarrow \partial_j(\tau)$ (the definition of $(\widehat{q, i, j})$ -nearness) implies $\partial_i(\sigma) = \partial_j(\tau) = \alpha$ exactly when $\sigma, \tau \in \Sigma_{q+1}$. Then, both $\partial_i(\sigma)$ and α are part of Σ_q , which turns the inclusion $\partial_i(\sigma) \hookleftarrow \alpha$ into an equivalence (and analogously for $\partial_j(\tau) = \alpha$).

A visual version of this proof can be seen in Figure 3.1. □

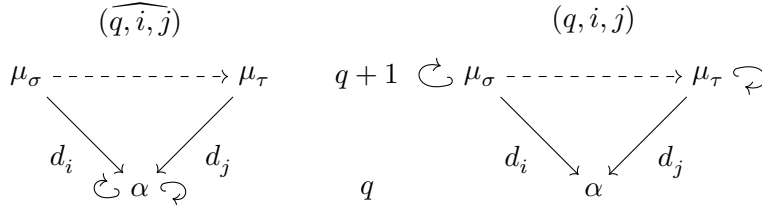


Figure 3.1.: Modified version of definition in Figure 2.2a making use of the reflexivity of inclusion shows that both definitions are equivalent for $(q + 1)$ -simplices.

The definitions also coincide for particularly important pairs of i and j :

Proposition 3.1.2 (0- ∞ -Equivalence). *Let G be a directed, simple graph with associated $(q, 0, \infty)$ -digraph \mathcal{Q} and $(q, 0, \infty)$ -digraph $\widehat{\mathcal{Q}}$. Then, $\mathcal{Q} = \widehat{\mathcal{Q}}$.*

This also holds true for the directions $(q, 0, 0)$, $(q, \infty, 0)$ and (q, ∞, ∞) .

Proof. All edges between simplices that fulfill criterion [I] are trivially contained in both \mathcal{Q} and $\widehat{\mathcal{Q}}$. It is sufficient to proof the statement for simplices that are near by criterion [II]:

$\widehat{Q} \subseteq Q :$

Assume two simplices $\sigma = (\sigma_0 \sigma_1 \cdots \sigma_n)$ and $\tau = (\tau_0 \tau_1 \cdots \tau_m)$ are $(q, \widehat{0}, \infty)$ -near. Then by definition there exists an $\alpha \in \Sigma_q$ such that $\widehat{\partial}_0(\sigma) \hookleftarrow \alpha \hookrightarrow \widehat{\partial}_\infty(\tau)$. Consider the faces $\mu_\sigma = (\sigma_0 \alpha_0 \alpha_1 \cdots \alpha_q) \hookrightarrow \sigma$ and $\mu_\tau = (\alpha_0 \alpha_1 \cdots \alpha_q \tau_m) \hookrightarrow \tau$, which share the (q) -face α . From $\sigma \hookleftarrow \widehat{\partial}_0(\mu_\sigma) = \alpha = \widehat{\partial}_\infty(\mu_\tau) \hookrightarrow \tau$, it follows that σ and τ are $(q, 0, \infty)$ -near. We note that μ_σ and μ_τ are well defined and do not contain duplicate vertices ($\sigma_0 \not\hookrightarrow \alpha \not\hookrightarrow \tau_m$), because $\alpha \hookrightarrow \widehat{\partial}_0(\sigma) = (\sigma_1, \cdots \sigma_n)$ and $\alpha \hookrightarrow \widehat{\partial}_\infty(\tau) = (\tau_0 \cdots \tau_{n-1})$.

$Q \subseteq \widehat{Q} :$

Consider two $(q, 0, \infty)$ -near simplices $\sigma = (\sigma_0 \sigma_1 \cdots \sigma_n)$ and $\tau = (\tau_0 \tau_1 \cdots \tau_m)$. By definition, there are two faces $\mu_\sigma \hookrightarrow \sigma$ and $\mu_\tau \hookrightarrow \tau$, such that $\partial_0(\mu_\sigma) = \partial_\infty(\mu_\tau) = \alpha$. Note that $\sigma_0 \notin \partial_0(\mu_\sigma)$ and $\tau_m \notin \partial_\infty(\mu_\tau)$, meaning neither can be contained in α . From this and $\sigma \hookleftarrow \alpha \hookrightarrow \tau$, we can conclude that $\partial_0(\sigma) \hookleftarrow \alpha \hookrightarrow \partial_\infty(\tau)$ and that σ, τ are $(q, \widehat{0}, \infty)$ -near

The proof for the three additional directions is analogous with different indices. For all other directions this proof does not hold, because it assumes that $\sigma_i = (\alpha \cup \sigma_i)_i$, where $\alpha \hookrightarrow \partial_i(\sigma)$. This is only guaranteed if $i = 0$ or $i = \infty$ (and identically for j).

□

Proposition 3.1.3. *If two simplices σ and τ are (q, i, j) -near, they are also (q, \widehat{k}, l) -near for some $k, l \in \mathbb{N}$ and vice versa.*

Proof. Should $\sigma \hookrightarrow \tau$ or $\sigma \hookleftarrow \tau$ hold true, both definitions trivially coincide for any i and j . Otherwise, both $(\sigma \setminus \tau) \neq \emptyset$ and $(\tau \setminus \sigma) \neq \emptyset$ hold and they must share a (q) -face α .

" \rightarrow " Consider the case where σ is (q, i, j) -near to τ . Let k be the index of any vertex $\sigma_k \in (\sigma \setminus \tau)$ in σ and l the index of any $\tau_l \in (\tau \setminus \sigma)$ in τ . From $\alpha \hookrightarrow \sigma$ and $\sigma_k \notin \alpha$ follows that $\alpha \hookrightarrow \partial_k(\sigma)$. Similarly, $\alpha \hookrightarrow \tau$ and $\tau_l \notin \alpha$ imply $\alpha \hookrightarrow \partial_l(\tau)$. The definition of (q, \widehat{k}, l) -near is fulfilled, since $\partial_k(\sigma) \hookleftarrow \alpha \hookrightarrow \partial_l(\tau)$

" \leftarrow " Consider the case where σ is (q, \widehat{i}, j) -near to τ . Let $\mu_\sigma = \alpha \cup v$ with $v \in (\sigma \setminus \tau)$ and $\mu_\tau = \alpha \cup w$ with $w \in (\tau \setminus \sigma)$. We choose k as the index of v in μ_σ and l as the index of w in μ_τ . Then, σ is (q, k, l) -near to τ , because $\partial_k(\mu_\sigma) = \partial_l(\mu_\tau) = \alpha$.

□

Note that Proposition 3.1.3 does not imply equivalence between the two definitions, as the values of k and l may be different for various (q, i, j) -near simplices.

In fact, for any simplices $\sigma, \tau \in \Sigma_{>q}$ that share a (q) -face there are indices i, j such that they are (q, i, j) -near. If two simplices share a $(q+2)$ -face, they are even (q, i, j) -near and (q, \widehat{i}, j) -near for any i, j :

Proposition 3.1.4 (Shared Face). *If the simplices σ, τ share a*

- *(q) -face, then there exist i and j , such that σ is (q, i, j) -near to τ .*
- *$(q+1)$ -face, then for all $i \in \mathbb{N}$, there exists some $j \in \mathbb{N}$, such that σ is (q, i, j) -near to τ (and vice versa there exists some i for all j).*
- *$(q+2)$ -face, then for all $i, j \in \mathbb{N}$, σ is (q, i, j) -near to τ .*

The same holds also for $(\widehat{q, i, j})$ -nearness.

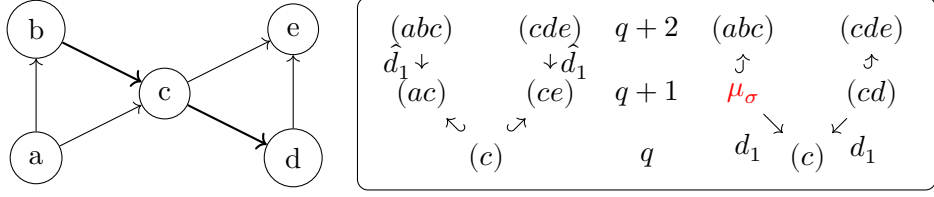
Proof.

- (q) -face:
If $\sigma \hookrightarrow \tau$ or $\tau \hookrightarrow \sigma$, they are trivially near by criterion [I]. Otherwise $\sigma \setminus \tau \neq \emptyset$ and $\tau \setminus \sigma \neq \emptyset$ hold. Choose i as the index of any $\sigma_i \in (\sigma \setminus \tau)$ and j as the index of any $\tau_j \in (\tau \setminus \sigma)$. For the chosen indices, the simplices are $(\widehat{q, i, j})$ -near (and by Proposition 3.1.3, there also exist indices j, l such that σ and τ are (q, i, j) -near).
- $(q+1)$ -face:
For any given i , $\partial_i(\sigma)$ and τ share a (q) -face α , because only one vertex is removed from σ . Choose j as the index of any $\tau_j \in (\tau \setminus \alpha)$. Such a τ_j must exist, because $\tau \in \Sigma_{>q}$ by assumption. For the chosen indices, the simplices are $(\widehat{q, i, j})$ -near (and by Proposition 3.1.3, there also exist indices j, l such that σ and τ are (q, i, j) -near).
- $(q+2)$ -face:
The simplices $\partial_i(\sigma)$ and $\partial_j(\tau)$ share at least a (q) -face, as at most two vertices (σ_i and τ_j) of the original $(q+2)$ -face are removed. Thus, the conditions for $(\widehat{q, i, j})$ -nearness are fulfilled. For (q, i, j) -nearness, assume w.l.o.g. that $i \leq j$. Choose μ_σ as $\partial_{j-1}(\alpha)$ and μ_τ as $\partial_i(\alpha)$. Then, because $\partial_i(\mu_\sigma) = \partial_j(\mu_\tau)$, criterion [II] is fulfilled.

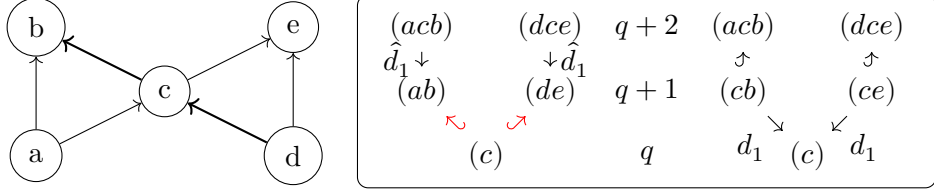
□

3.2. Differences

While both definitions share many traits, they are not equivalent. Simplices may be $(\widehat{q, i, j})$ -near, but not (q, i, j) -near and vice versa.



(a) The simplex (abc) is $(0, \widehat{1, 1})$ -near to (cde) , but not $(0, 1, 1)$ -near



(b) The simplex (acb) is $(0, 1, 1)$ -near to (dce) , but not $(0, \widehat{1, 1})$ -near.

Example 3.2.1. Consider the two ordered simplices (abc) and (cde) in the graph of Figure 3.2a. They are $(0, \widehat{1, 1})$ -near, because $\hat{\partial}_1(abc) \hookleftarrow (c) \hookrightarrow \hat{\partial}_1(cde)$. The only candidate for α in this example is (c) , as it is the only shared vertex. In the new definition, (abc) and (cde) are not $(0, 1, 1)$ -near, as there is no $\mu_\sigma \hookrightarrow (abc)$ such that $\partial_1(\mu_\sigma) = (c)$.

Conversely, the two simplices (acb) and (dce) in the slightly modified graph from Figure 3.2b are $(0, 1, 1)$ -near, but not $(0, \widehat{1, 1})$ -near since there is no α , such that $\hat{\partial}_1(acb) = ab \hookleftarrow \alpha \hookrightarrow (de) = \hat{\partial}_1(dce)$. Under which circumstances the two definitions diverge is explored further in Sections 3.3 and 3.4.

The following property of (q, i, j) -nearness is particularly useful in the construction of efficient algorithms, as exploited in section 4.2.4.

Lemma 3.2.2 (Upward Closure). *For any two (q, i, j) -near $(q + 1)$ -simplices $\mu_\sigma \hookrightarrow \sigma$ and $\mu_\tau \hookrightarrow \tau$, it holds that σ is also (q, i, j) -near to τ .*

Proof. This follows directly from the definition of (q, i, j) -nearness. Since $\mu_\sigma, \mu_\tau \in \Sigma_{q+1}$, they share the face $\alpha = \partial_i(\mu_\sigma) = \partial_j(\mu_\tau)$ (see Figure 3.1). The (q, i, j) -nearness follows from $\mu_\sigma \hookrightarrow \sigma$ and $\mu_\tau \hookrightarrow \tau$. \square

Note that this proof also works if μ_σ, μ_τ are of higher dimension than $(q + 1)$.

Corollary 3.2.3. *Because every simplex is contained in itself ($\sigma \hookrightarrow \sigma$), if any σ is (q, i, j) -near to τ , then it is also (q, i, j) -near to all $\tau' \hookleftarrow \tau$.*

Remark 3.2.4. Lemma 3.2.2 does not hold for $(\widehat{q, i, j})$ -nearness. Consider the subset of a flag complex in Figure 3.3. The red edges are contained in the $(2, 1, 3)$ -near-digraph, but not in the $(2, 1, 3)$ -near-digraph. According to Lemma 3.2.2, if $\sigma \dashrightarrow \tau$, then $\sigma' \dashrightarrow \tau'$ should hold. While σ is $(2, 1, 3)$ -near to τ , this is not the case for σ' and τ' , because $\partial_1(\sigma') = (0234)$ and $\partial_3(\tau') = (0135)$ do not share a (2) -face.

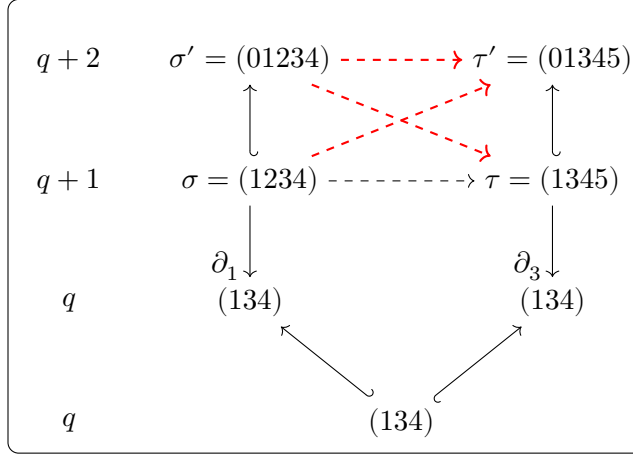


Figure 3.3.: Subset of flag complex that represents a counterexample to Lemma 3.2.2 for $(\widehat{q, i, j})$ -nearness.

We now show the equivalence of the definition for (q, i, j) -nearness used in the introduction and the formal definition from Section 2.

Lemma 3.2.5. *A simplex $\sigma \in \Sigma_{>q}$ is (q, i, j) -near to any simplex τ iff at least one $(q+1)$ -simplex $\mu_\sigma \hookrightarrow \sigma$ is (q, i, j) -near to τ .*

Proof. We show both directions separately:

” \rightarrow ” If σ is (q, i, j) -near to τ , then by definition there exist $(q+1)$ -simplices $\mu_\sigma \hookrightarrow \sigma$, $\mu_\tau \hookrightarrow \tau$ such that $\partial_i(\mu_\sigma) = \partial_j(\mu_\tau) = \alpha$. The face μ_σ is then also (q, i, j) -near to τ , because $\mu_\sigma \hookrightarrow \mu_\sigma$.

” \leftarrow ” follows from Lemma 3.2.2.

□

Lemma 3.2.5 can be understood as the reverse counterpart of Lemma 3.2.2.

A major difference between the two definitions is the role of the indices i and j . When considering the $(\widehat{q, i, j})$ -nearness of σ and τ , indices i and j will be used to refer to σ_i and τ_j respectively. In the new definition, i and j are indices for subsimplices $\mu_\sigma, \mu_\tau \in \Sigma_{q+1}$, where $\mu_{\sigma_i} = \sigma_{\tilde{i}}$. The index \tilde{i} then falls somewhere in the range of $[i, \dim(\sigma) - q + i - 1]$. This interval originates from the fact that i vertices of μ_σ occur before $\sigma_{\tilde{i}}$ and the other $q+1-i$ vertices of μ_σ after $\sigma_{\tilde{i}}$.

At first glance, it would appear that this relaxation would lead to a less strict definition, but there is another hidden assumption at play.

Theorem 3.2.6 (Split Index). *Let $\sigma \in \Sigma_n$ and $\tau \in \Sigma_m$. The simplices σ and τ are (q, i, j) -near iff there exists $i' \geq i$, which decomposes σ into $(\sigma_{\triangleleft}, \sigma_{i'}, \sigma_{\triangleright})$ and $j' \geq j$, which decomposes τ into $(\tau_{\triangleleft}, \tau_{j'}, \tau_{\triangleright})$ such that*

- σ_{\triangleleft} and τ share an $(i-1)$ -face $\hat{\sigma}_{\triangleleft}$,
- σ_{\triangleright} and τ share a $(q-i)$ -face $\hat{\sigma}_{\triangleright}$,
- τ_{\triangleleft} and σ share a $(j-1)$ -face $\hat{\tau}_{\triangleleft}$,
- τ_{\triangleright} and σ share a $(q-j)$ -face $\hat{\tau}_{\triangleright}$ and
- $(\hat{\sigma}_{\triangleleft}, \hat{\sigma}_{\triangleright}) = (\hat{\tau}_{\triangleleft}, \hat{\tau}_{\triangleright})$.

Proof. We show both directions separately:

" \leftarrow " Assume there exist indices i', j' as described in the theorem for some simplices σ, τ . Then a shared face α can be constructed as $(\sigma_{\triangleleft} \cap \tau) \cup (\sigma_{\triangleright} \cap \tau)$. The first intersection will contain i vertices and the second $(q-i+1)$ vertices due to the shared $(i-1)$ - and $(q-i)$ -face respectively. Thus, $\dim(\alpha) = (i + (q-i+1)) - 1 = q$. Then $\alpha = \partial_i((\alpha_0, \dots, \alpha_{i-1}, \sigma_{i'}, \alpha_i, \dots, \alpha_q)) = \partial_i(\mu_\sigma) \hookrightarrow \sigma$. The same construction to find μ_τ completes this direction and the last condition guarantees that $\partial_i(\mu_\sigma) = \partial_j(\mu_\tau)$.

" \rightarrow " Let σ and τ be (q, i, j) -near. By definition, there exist $(q+1)$ -simplices $\mu_\sigma \hookrightarrow \sigma, \mu_\tau \hookrightarrow \tau$ such that $\partial_i(\mu_\sigma) = \alpha = \partial_j(\mu_\tau)$ for some $\alpha \in \Sigma_q$. Choose i' as the lowest possible value such that $\mu_{\triangleleft} = (\mu_{\sigma_0} \mu_{\sigma_1} \dots \mu_{\sigma_{i-1}})$ is still contained in σ_{\triangleleft} . By definition of (q, i, j) -nearness, $\mu_{\triangleleft} \hookrightarrow \alpha$, and since $\alpha \hookrightarrow \tau$ and α contains i vertices of σ_{\triangleleft} , the simplex σ_{\triangleleft} shares a $(i-1)$ -face with τ . Now σ_{\triangleright} must contain all vertices of $\mu_{\triangleright} = (\mu_{\sigma_{i+1}} \mu_{\sigma_{i+2}} \dots \mu_{\sigma_n})$, which are in turn contained in α and thus also in τ , making μ_{\triangleright} a shared $(q-i)$ -face between σ_{\triangleright} and τ . The other conditions follow by a symmetric argument for τ .

□

Example 3.2.7. Consider the two simplices $\sigma = (014567)$ and $\tau = (12346)$, which are $(2, 2, 1)$ -near ($\partial_2((1456)) = \partial_1((1246))$). The indices can be chosen as $i' = 3$ and $j' = 2$. The simplex σ decomposes into $\sigma_{\triangleleft} = (014)$, $\sigma_{\triangleright} = (67)$ and τ into $\tau_{\triangleleft} = (12)$, $\tau_{\triangleright} = (46)$. We observe that (014) shares the $(i-1) = (1)$ -face (14) with τ and the $(q-i) = (0)$ -face (6) is shared between σ_{\triangleright} and τ . Conversely, σ shares face (1) with τ_{\triangleleft} and (46) with τ_{\triangleright} , satisfying the conditions from Theorem 3.2.6.

Example 3.2.8. The fact that Theorem 3.2.6 does not hold for $(\widehat{q, i, j})$ -nearness is evident from Example 3.2.1. For the simplices (abc) and (cde) (see Figure 3.2a), no indices $i' \geq 1$ and $j' \geq 1$ can be found that fulfill the conditions, but they are still $(0, 1, 1)$ -near. Conversely, the simplices (acb) and (dce) (see Figure 3.2b) are not $(0, 1, 1)$ -near, even though for $i' = 2$ and $j' = 2$ they can be shown to be $(0, 1, 1)$ -near. A (-1) -face in this case represents the empty set.

We observe that while i and j in the original definition are "fixed" to particular vertices, the number of shared vertices before or after the i th or j th vertex does not impact $(\widehat{q, i, j})$ -nearness. With the new definition, i and j may refer to different vertices in the same simplex, but under the additional constraint that at least $i - 1$ resp. $j - 1$ shared vertices occur in order before that vertex. The definitions can thus be seen as having a different degree of freedom. Interestingly, while small differences occur in practice, no definition can be considered more strict than the other, as will be shown in the next section.

3.3. Combinatorial Analysis

Viewing directed Q-analysis from a combinatorial standpoint raises a number of interesting questions. How many possible combinations of (q, i, j) -near or $(\widehat{q, i, j})$ -near simplices exist? How many of them fulfill both the original and new definition? In particular, how is this affected by the dimension of the simplices or the values of i and j ?

3.3.1. Counting

All simplex pairs that are (q, i, j) -near can be represented as directed graphs, meaning they can be enumerated and counted. In this context, it makes sense to restrict the analysis to only oriented graphs that form a pair of simplices, which have a common face α (which is then guaranteed to be a directed simplex, as there are no bidirectional edges in an oriented graph). Note that simplex pairs that share less than $(q + 1)$ vertices can never be (q, i, j) -near or $(\widehat{q, i, j})$ -near and pairs that share more than $(q + 2)$ vertices are always near by both definitions (recall Proposition 3.1.4 about shared faces). We will focus in this analysis only on the most interesting case of shared (q) -faces, but the result that both definitions are equally strict is equally applicable for shared $(q + 1)$ -faces.

The number of possible simplex pairs $(\sigma, \tau) \in \Sigma_n \times \Sigma_m$ that share a (q) -face can be expressed as $\binom{n+1}{q+1} \cdot \binom{m+1}{q+1}$. The binomial coefficients count all the possible ways to pick a $(q + 1)$ -vertex subgraph from σ and τ respectively. Among those possible pairs, the number of pairs that are $(\widehat{q, i, j})$ -near is:

$$\hat{\#}(n, m, q) = \binom{n}{q+1} \cdot \binom{m}{q+1}.$$

Here, the binomial coefficients count the ways that the $q + 1$ vertices of α can be chosen in $\partial_i(\sigma)$ and $\partial_j(\tau)$ respectively. The number of $(\widehat{q, i, j})$ -near simplex pairs is thus the

same for any pair i, j .

Surprisingly, the exact same formula also holds for (q, i, j) -nearness, although it is not obvious. First, we consider just the possible subsets α of one simplex σ that could lead to (q, i, j) -nearness (equivalent to the first binomial coefficient in previous formula). The Split Index Theorem 3.2.6 can be used to enumerate all possible values for α . If σ, τ are (q, i, j) -near, then for every possible value of the split index i' , exactly i shared vertices must occur in the first $i' - 1$ vertices of σ and $q - i + 1$ shared vertices are found in the rest of the σ . This also limits the legal values for the split index to the range of $[i + 1, n - q + i]$. Summing the possible combinations of faces $\mu_{\triangleleft} \subseteq \sigma_{\triangleleft}$ and $\mu_{\triangleright} \subseteq \sigma_{\triangleright}$ over i' results in:

$$\sum_{i'=i+1}^{n-q+i} \binom{i'-1}{i} \cdot \binom{n-i'+1}{q-i+1}$$

However, there is still some overcounting occuring as certain α may be counted for multiple values of i' . In order to remedy this, we can count for each value of i' only the faces α that contain the vertex immediately ahead of $\sigma_{i'}$ ($\sigma_{i'-1} \in \alpha$). As this vertex of α is now fixed, only $i - 1$ vertices are picked from the first part of σ_{\triangleleft} (which is also shortened by the fixed vertex) instead:

$$\sum_{i'=i+1}^{n-q+i} \binom{i'-2}{i-1} \cdot \binom{n-i'+1}{q-i+1}$$

For $1 \leq i \leq q + 1$, this is exactly equal to $\binom{n}{q+1}$ in the range of $i, n, q \in [1, 100]$ (and probably outside as well). Similarly, there are $\binom{m}{q+1}$ ways to pick $q + 1$ vertices from τ such that σ and τ can be (q, i, j) -near. As such, for any directions (q, i, j) there are the same number of combinations of near simplex pairs for both definitions:

$$\hat{\#}(n, m, q) = \#(n, m, q) = \binom{n}{q+1} \cdot \binom{m}{q+1}.$$

From this we can conclude that no definition or index pair can be considered "stricter" than another. Note that this is not in conflict with the fact that the definitions are different: While the sizes of the sets may be identical, their elements are not for $0 < i, j < q + 1$. The question of how much the two sets overlap for different values of i and j naturally arises.

3.3.2. Enumerating

A way to tackle this question theoretically eludes us. Instead, we run a small comparative experiment. All aforementioned simplex pairs in oriented graphs for fixed n, m with shared (q) - and $(q + 1)$ -faces respectively were enumerated and then checked for both (q, i, j) -nearness and (q, i, j) -nearness.

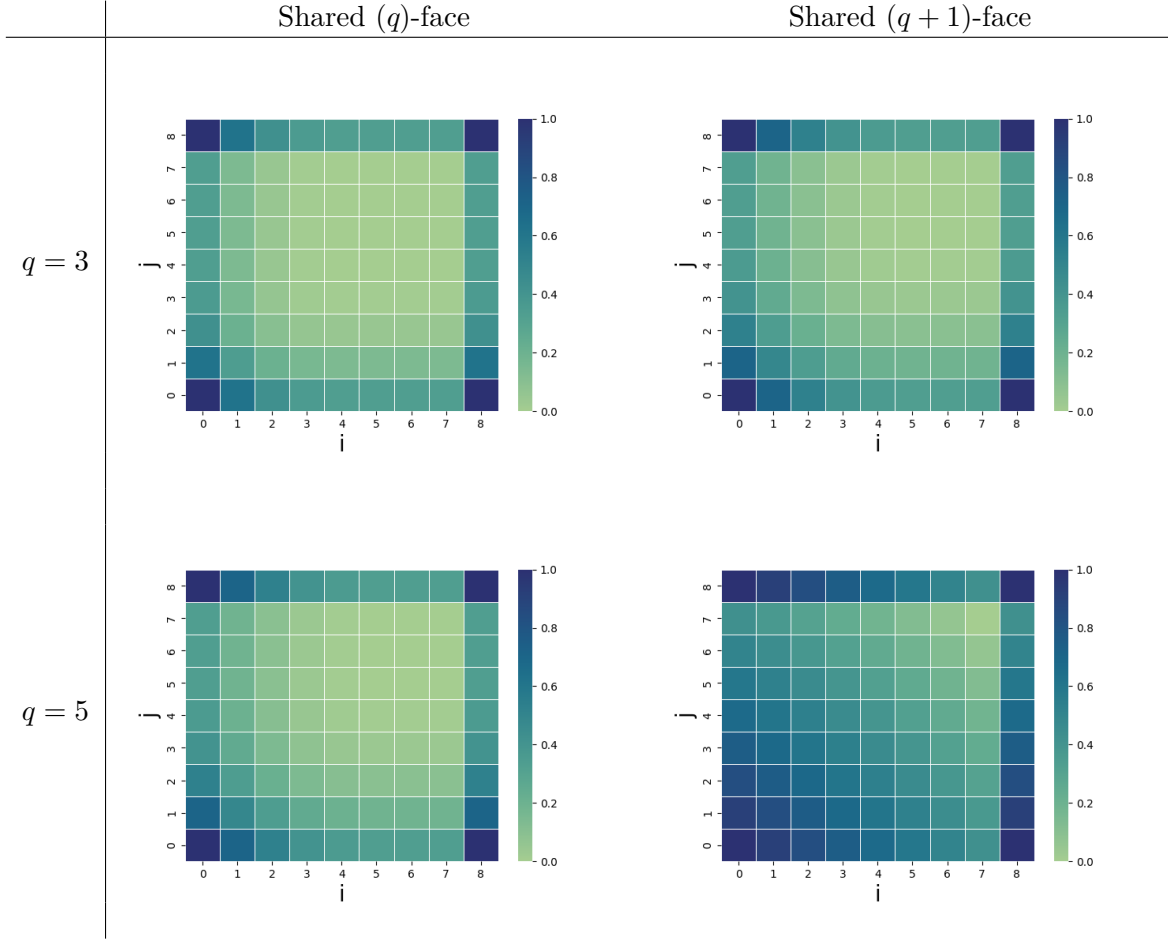


Table 3.1.: Heatmaps showing the percentage of (q, i, j) -near pairs of (7) -simplices (out of all possible pairs that share a (q) - or $(q + 1)$ -face) that are also $(\widehat{q, i, j})$ -near for different values of q , i , j and $\dim(\alpha)$.

The result for all combinations of 7-simplices can be seen in Table 3.1. In the corner cases, both definitions align perfectly. This is unsurprising due to Proposition 3.1.2 about 0 - ∞ -equivalence. Note that the direction $(q, \dim(\sigma) + 1, \dim(\tau) + 1)$ in the top right corners is equivalent to (q, ∞, ∞) for both definitions. The figure illustrates the interesting pattern that the definitions continue to diverge as i and j increase, but then perfectly align in the extreme. It can also be seen that the definitions diverge more quickly for lower values of q , but are more aligned when sharing a $(q + 1)$ -face. The second fact can be explained due to the definitions being generally looser (as explored in Proposition 3.1.4) in that case and thus also creating more overlap.

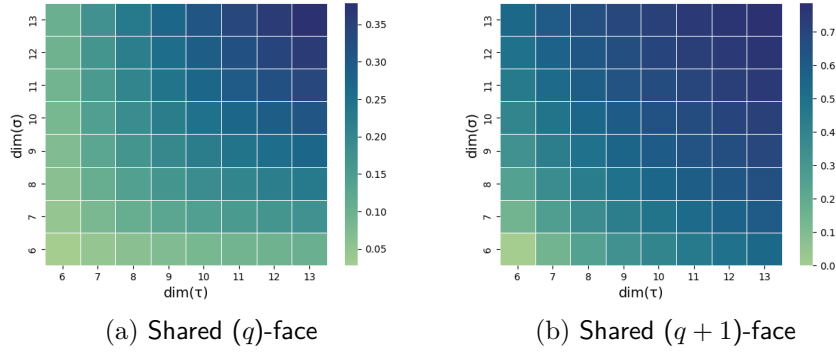


Figure 3.4.: Heatmap showing the proportion of simplex pairs that share a (4) - / (5) -face and are $(4, i, j)$ -near.

The question remains how the simplex dimensions $n = \dim(\sigma)$ and $m = \dim(\tau)$ affect the likelihood of two simplices being (q, i, j) -near. The answer can be found in Figure 3.4, which shows the proportion of simplex pairs that are $(4, i, j)$ -near with $6 \leq \dim(\sigma), \dim(\tau) \leq 13$. As shown in the previous section, the results are equivalent for both the new and original definition and all values of i, j . It can be seen that simplices of higher dimension are significantly more likely to be (q, i, j) -near or (q, i, j) -near. The explanation for this becomes apparent when examining the definition of (q, i, j) -nearness. Two simplices that share a (q) -face α are (q, i, j) -near iff $\sigma_i \notin \alpha, \tau_j \notin \alpha$. If the size of σ and τ increases, while $\dim(\alpha)$ remains the same, the probability that σ_i or τ_j is “picked” to be in α shrinks. From another perspective, the degrees of freedom mentioned in the previous section for both definitions increase with the dimension of the simplices. Unsurprisingly, simplices are more likely to be near when they share a $(q+1)$ -face, which is a direct result of the Shared Face Proposition 3.1.4. The source code for generating these diagrams is included in ².

3.4. Visualization

Figure 1.1a shows how undirected q -nearness relations can be effectively visualized using Venn diagrams. For directed q -nearness, more sophisticated methods are required to present the concepts in an approachable manner. This section will present various methods to visualize the concept of directed q -nearness and build intuition.

Bipartite Graph Drawings

Any two directed simplices σ, τ that share a face can be mapped to a unique bipartite graph $G_b = (V_1, V_2, E)$ with $V_1 = \{\sigma_0, \dots, \sigma_n\}, V_2 = \{\tau_0, \dots, \tau_m\}$ and $E = \{(s, t) \mid s \in \sigma \wedge t \in \tau \wedge s = t\}$. Figure 3.5 shows a straight-line drawing of the bipartite graph with the vertices of σ in order above the ordered vertices of τ . The simplices are then for

²https://github.com/FelixWindisch/DirQ/blob/main/supplementals/q_enumeration.py

example $(\widehat{q, i, j})$ -near if a subset of $q + 1$ edges can be found in G_b that do not include σ_i or τ_j and are crossing-free in the aforementioned straight-line drawing.

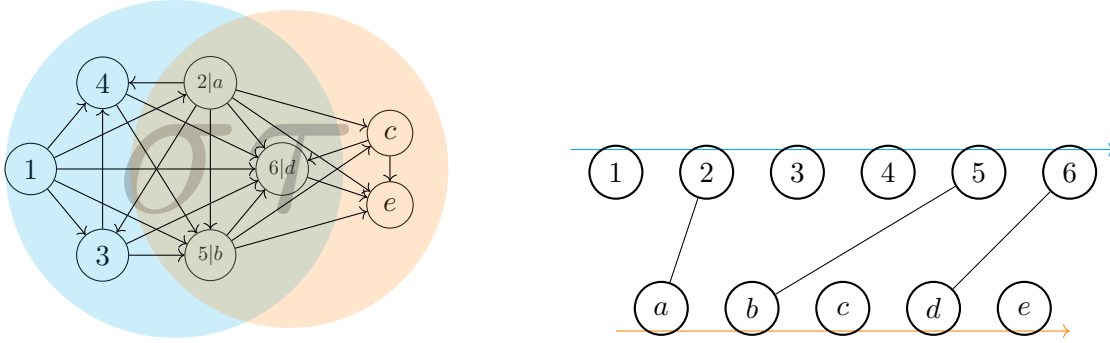
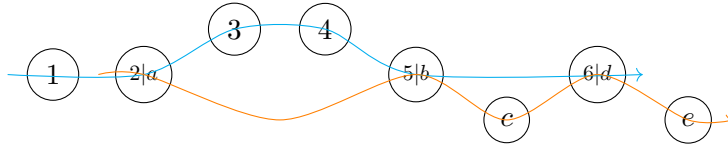


Figure 3.5.: Two directed simplices sharing a face and the corresponding bipartite graph drawing.

Arrow Diagrams

When drawing bipartite graphs for directed simplices in this way, two parallel arrows through the top and bottom row can be understood as representing the “flow” that each simplex represents. If the vertices are merged along edges, the respective arrows will coincide wherever vertices are shared or run in parallel otherwise. The following *arrow diagram* depicts the simplex pair in Figure 3.5:



These new methods of visualizations give better insight into the differences of the two definitions. In order to make sense of the directionality, we revisit the pairs of simplices that share a (q) -face in an oriented graph, as described in the previous section. Some examples of arrow diagrams for $(2, 1, 1)$ -near and/or $(\widehat{2, 1, 1})$ -near (4) -simplex pairs are displayed in Figure 3.6. Upon inspection, certain patterns emerge from these arrow diagrams. For the $(\widehat{2, 1, 1})$ -near pairs (σ, τ) , that pattern is simply that σ_i and τ_j (marked in gray) are not part of the shared simplex. The $(2, 1, 1)$ -near simplex pairs all contain a specific structure that is highlighted in white on the bottom right of the figure, which is not found in the pairs that are only $(\widehat{2, 1, 1})$ -near. The center of Figure 3.6 consists of examples of simplex pairs that exhibit both patterns and are thus near by both definitions.

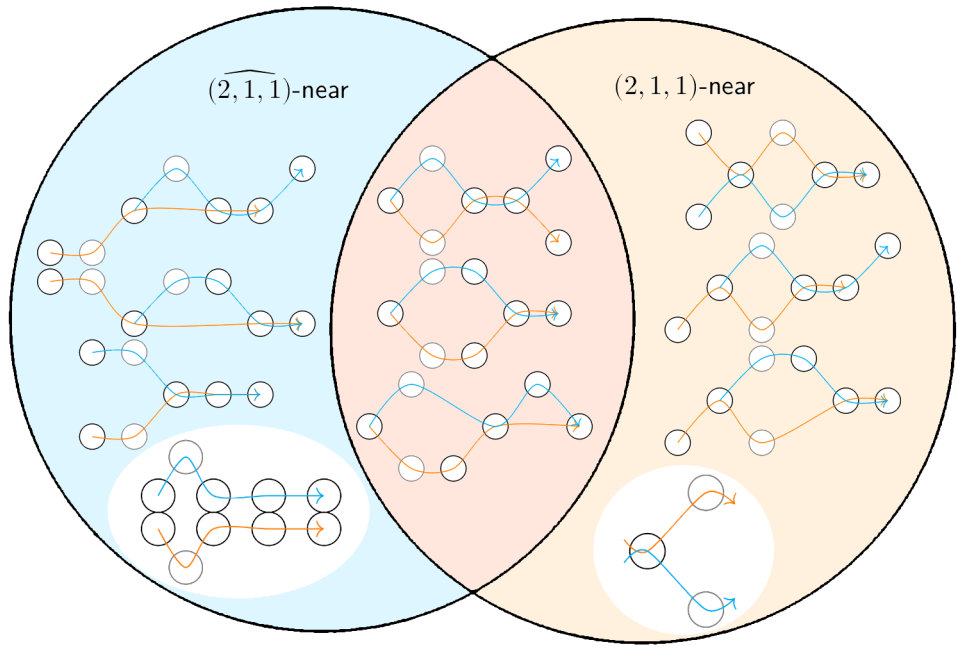


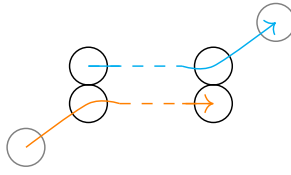
Figure 3.6.: Arrow diagrams of simplex pairs that are $(\widehat{2,1,1})$ -near (left), $(2,1,1)$ -near (right) or both (middle). The structure of the $(2,1,1)$ direction for both definitions is highlighted in white.

Identifying nearness in arrow diagrams

With the example from Figure 3.6 in mind, the following details how to identify nearness of both definitions in arrow diagrams.

Simplex pairs that share less than $q + 1$ vertices in the same order are trivially not q -near by both definitions, while those that share more than $q + 2$ are always (q, i, j) -near for all i, j and definitions. For parallel pairs that share $(q + 1)$ vertices (resp. $(q + 2)$ vertices)), the following criteria can be applied:

Original Definition The two simplices are (q, i, j) -near iff the i th vertex of σ and (resp. or) the j th vertex of τ are not shared. For example, all $(q, 0, \infty)$ -near pairs that share a (q) -simplex will always follow this pattern:



New Definition Identifying (q, i, j) -nearness is more complicated. As observed in Figure 3.6, all pairs that are $(2, 1, 1)$ -near exhibit a specific structure. This is a direct result from the definition of faces μ_σ and μ_τ , which share all vertices except the i th and (resp. or) j th respectively. In the example, $i = j = 1$, so the first vertex of μ_σ and μ_τ must be shared and the second must not be (if $\dim(\sigma) = \dim(\tau) = q + 1$), leading to the particular structure. Note that this structure marks the beginning of μ_σ resp. μ_τ , and no shared vertices may occur before it. Figure 3.7 compiles these particular structures for different pairs of i and j and shared (q) -simplices. Dotted lines act as a placeholder for an arbitrary number of non-shared vertices. These reflect the fact that any number of vertices of σ may be between two vertices of the subsimplex μ_σ (similarly for τ and μ_τ). Observe also that the structure of $(q, 0, 0)$ -nearness is also contained in every pair of $(q, 0, 0)$ -near simplices, giving some visual intuition for the 0- ∞ -Equivalence Proposition 3.1.2. When the two simplices share a $(q + 1)$ -face, the pattern is less strict: Only one of the two involved simplices has to follow the structure.

Summary As shown in Proposition 3.1.2, the definitions are equivalent for many important directions (q, i, j) . As the definitions diverge for higher $i, j < (q + 1)$, the visualizations tell us something important about the priorities of the definitions:

- (q, i, j) -nearness tells us about *which parts* of the simplices intersect. In particular, whether σ_i or τ_j are involved in the intersection.
- (q, i, j) -nearness tells us *how* the simplices intersect. The intersection and not the positions are characterized (see Figure 3.7).

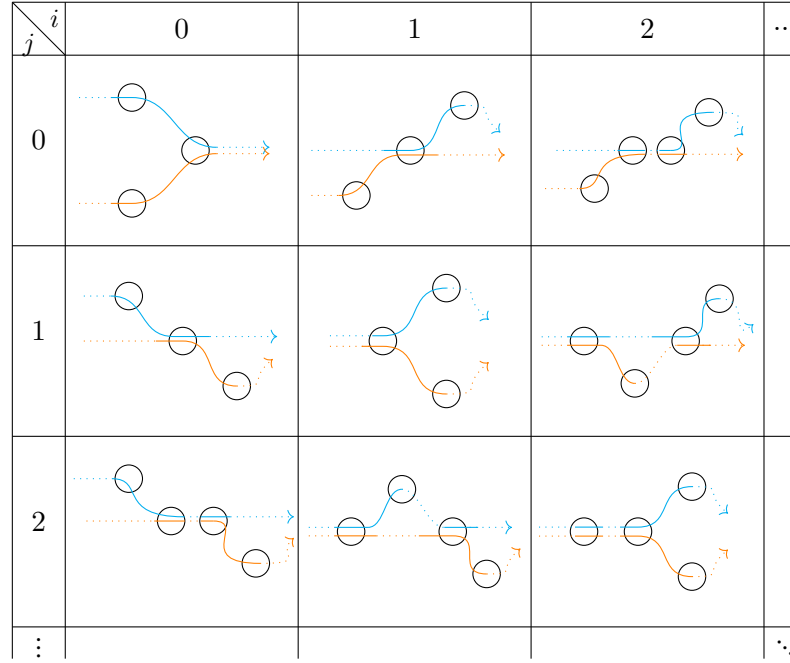
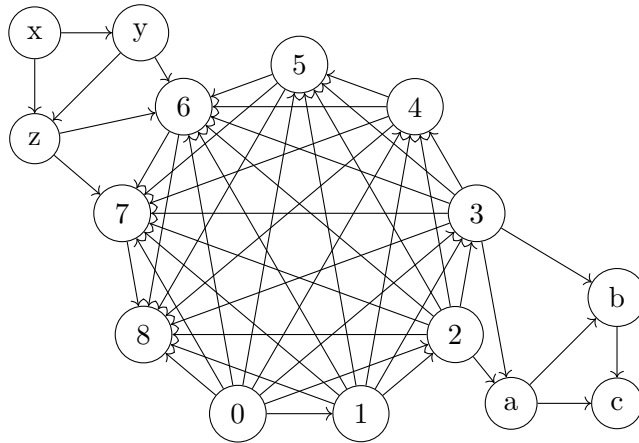


Figure 3.7.: Arrow diagrams for structures that are contained in (q, i, j) -near simplex pairs with shared (q) -face.

3.5. Limiting Simplex Dimension

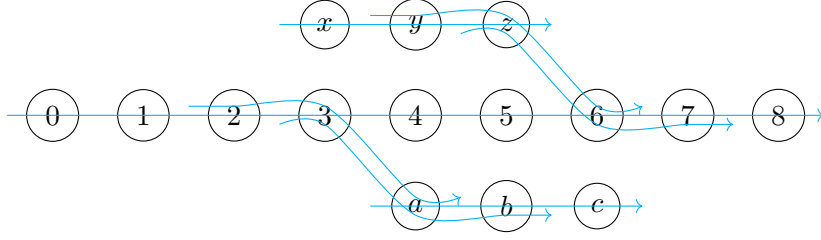
Whenever simplices of a dimension significantly higher than q occur, both definitions for directed q -nearness can lead to unintuitive results.

Example 3.5.1. One such case is depicted in the following graph:



The graph is composed of two smaller simplices that are connected on either sides to a large, central simplex. The direction of flow of the central simplex points from the

intersection with (abc) towards the intersection with simplex (xyz) , as can be seen in the following arrow diagram:



As seen in previous examples, the direction $i = 0, j = \infty$ is typically characterized by continued forward flow. Unintuitively, the $(1, 0, \infty)$ -digraph in this example contains the following path from (xyz) to (abc) for both definitions, seemingly traversing the central simplex against the direction of flow:

$$(xyz) \dashrightarrow (yz6) \dashrightarrow (z67) \dashrightarrow (678) \hookrightarrow (12345678) \dashrightarrow (23a) \dashrightarrow (3ab) \dashrightarrow (abc)$$

The surprising result originates specifically from the inclusion edge $(678) \hookrightarrow (12345678)$. As criterion [I] is not concerned with the direction of simplices, it enables paths that traverse against the direction of the underlying graph through sufficiently large simplices.

These kind of cases are common, whenever simplices with dimension significantly larger than q occur. The combinatorial analysis has also shown that for simplices of greater dimension, the definitions of directed q -nearness are less strict. In a sense, larger simplices act as "hubs" in the (q, i, j) -digraph, but if the dimension of simplices is too far away from q , the directionality is weakened.

Therefore, we propose to limit the dimension of simplices, when analyzing directed graphs. That is, reducing the vertex set of \mathcal{Q} from Σ to $\Sigma_{\leq M}$ for some maximal dimension M somewhere in the range of $[q + 3, q + 6]$.

Remark 3.5.2. When limiting the dimension of simplices, $\Sigma_{\leq M}$ is no longer considered a directed flag complex, because it does not contain all directed simplices. Instead, it is just a *simplicial complex*. For the remainder of this thesis, we will still refer to the full flag complex Σ , but all presented algorithmic ideas are still applicable for $\Sigma_{\leq M}$.

4. Algorithms to Compute (q, i, j) -Digraphs

We present, given an input graph G , algorithms to efficiently compute either the (q, i, j) -near-digraph \mathcal{Q} or the (\widehat{q}, i, j) -near-digraph $\widehat{\mathcal{Q}}$. In practice, we assume the directed flag complex Σ of G to be our input, as it determines both \mathcal{Q} and $\widehat{\mathcal{Q}}$. The computation of Σ can be handled by established algorithms like **flagser** [Lü17]. In fact, only $\Sigma_{>q}$ is required, but it is not possible to skip the bottom layers in the computation.

Algorithmically, the problem of computing the (q, i, j) -digraph can be thought of as identifying all instances of the pattern shown in Figure 2.2a within the flag complex. Each instance of these patterns is then associated with one edge of \mathcal{Q} or $\widehat{\mathcal{Q}}$ (see Figure 4.1).

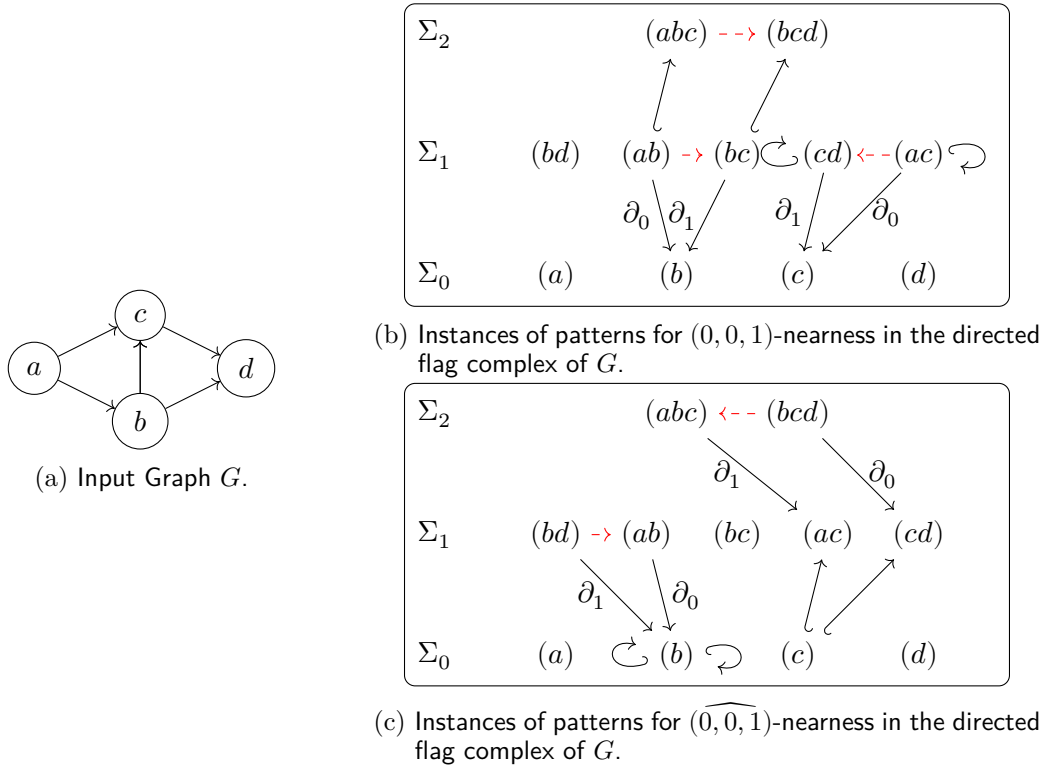


Figure 4.1.: The (q, i, j) -digraph can be computed by finding all instances of patterns from Figure 2.2a in the flag complex.

4.1. Naive Algorithm

The following Subsection formalizes the algorithmic ideas that were used to analyze graphs using directed q -nearness in Riihimäki [Rii23] and extends them to the novel definition.

The outline of the straightforward procedure to compute the (q, i, j) -digraph \mathcal{Q} is given in Algorithm 1. The method iterates over all edges that could be part of the (q, i, j) -digraph (all pairs of simplices in $\Sigma_{\geq q}$) and decides whether or not they form an edge in \mathcal{Q} .

Algorithm 1 Naive algorithm.

```

1: function GET_ $\mathcal{Q}$ _NAIVE( $\Sigma, q, i, j$ )
2:    $\mathcal{Q} \leftarrow \text{unconnected\_graph}(\Sigma_{\geq q})$ 
3:   for all  $(\sigma, \tau)$  in  $(\Sigma_{\geq q} \times \Sigma_{\geq q})$  do  $\triangleright [\mathcal{O}(\Sigma_{\geq q}^2)]$ 
4:     if IS_ $\mathcal{Q}$ _NEAR( $\sigma, \tau, q, i, j$ ) then
5:        $\mathcal{Q}.\text{add}(\sigma \rightarrow \tau)$   $\triangleright$  Add edge
6:     end if
7:   end for
8:   return  $\mathcal{Q}$ 
9: end function

```

Line 4 of Algorithm 1 may use the original or the new definition of directed q -nearness.

To decide that two arbitrary simplices σ, τ are $(\widehat{q, i, j})$ -near, either at least one (q) -face shared by $\partial_i(\sigma)$ and $\partial_j(\tau)$ has to be found or that $\sigma \hookrightarrow \tau$. Algorithm 2 determines $(\widehat{q, i, j})$ -nearness using the faces operator to enumerate the set of (q) -faces for σ and τ .

Algorithm 2 Check $(\widehat{q, i, j})$ -nearness of two simplices.

```

1: function IS_ $\widehat{\mathcal{Q}}$ _NEAR( $\sigma, \tau, q, i, j$ )
2:   if  $|\sigma \cap \tau| > q$  then
3:     return  $(|\text{faces}(\partial_i(\sigma), q) \cap \text{faces}(\partial_j(\tau), q)| > 0 \text{ or } \sigma \hookrightarrow \tau)$ 
4:   end if
5:   return false
6: end function

```

Two simplices σ, τ are (q, i, j) -near if there exist $(q + 1)$ -faces $\mu_\sigma \hookrightarrow \sigma, \mu_\tau \hookrightarrow \tau$ such that $\partial_i(\mu_\sigma) = \partial_j(\mu_\tau)$. As such, they are (q, i, j) -near if there is a non-empty intersection between the sets $\alpha_\sigma = \{\partial_i(\mu_\sigma) \mid \mu_\sigma \in \text{faces}(\sigma, q + 1)\}$ and $\alpha_\tau = \{\partial_j(\mu_\tau) \mid \mu_\tau \in \text{faces}(\tau, q + 1)\}$. This procedure is formalized in Algorithm 3.

Algorithm 3 Check (q, i, j) -nearness of two simplices.

```

1: function IS_Q_NEAR( $\sigma, \tau, q, i, j$ )
2:   if  $|\sigma \cap \tau| > q$  then
3:      $\alpha_\sigma \leftarrow [\text{faces}(\sigma, q+1).map(x \rightarrow \partial_i(x))]$ 
4:      $\alpha_\tau \leftarrow [\text{faces}(\tau, q+1).map(x \rightarrow \partial_j(x))]$ 
5:     return  $(|\alpha_\sigma \cap \alpha_\tau| > 0 \text{ or } \sigma \hookrightarrow \tau)$ 
6:   end if
7:   return false
8: end function

```

Correctness

Figure 4.2 gives a visual overview of how IS_Q_NEAR and IS_Q̂_NEAR identify criterion [II] for both definitions. The correctness of Algorithm 1 follows from the iteration over all pairs of vertices in \mathcal{Q} and checking them against the respective q -nearness definition.

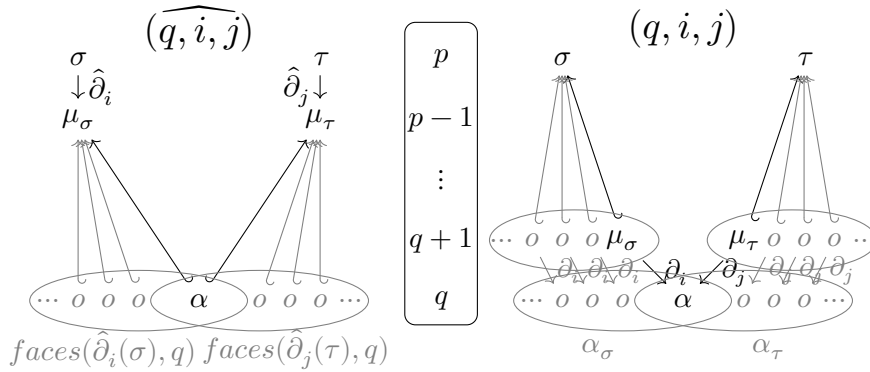


Figure 4.2.: Schematic of intersection tests in Algorithm 2 (left) and 3 (right).

Space Complexity

The space complexity of Algorithm 1 is dominated by the output \mathcal{Q} , which may contain $\mathcal{O}(|\Sigma_{\geq q}|^2)$ edges for a dense graph.

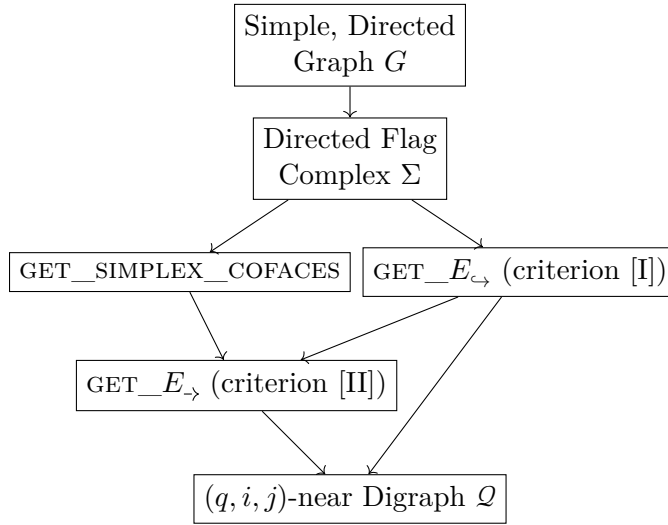
Time Complexity

The major downside of this algorithm is the quadratic $\mathcal{O}(|\Sigma_{\geq q}|^2)$ number of q -nearness checks that need to be performed even in the best case scenario. The time complexity of each check depends on q and the dimension of σ and τ . Giving an accurate analysis of this runtime is difficult, as it heavily depends on the structure of the graph. In practice, high dimensional simplices are rare and $\dim(\sigma)$ can reasonably be treated as a constant. The set intersections in Algorithms 2 and 3 may be computed in linear time with respect to simplex dimension by inserting all simplices into a hashmap and checking for collisions.

4.2. Improved Algorithm to Compute \mathcal{Q}

The above analysis hints towards some potential for optimization. The worst case complexity of $\mathcal{O}(|\Sigma_{\geq q}|^2)$ cannot be improved, as this is also the worst-case output size. However, it is possible to compute \mathcal{Q} using an output-sensitive algorithm that can avoid the high overhead of q -nearness checks at the cost of increased memory consumption. In the following, the maximal dimension of simplices D is treated as a constant, as it is either explicitly bounded (as described in Section 3.5) or very small in practice ($\dim(\sigma) \leq 25$ for all considered datasets).

The main idea of the algorithm is to first find edges of \mathcal{Q} that satisfy criterion [I] using a top-down approach (starting from higher dimensional simplices). Then, all (q, i, j) -near relations that follow criterion [II] are computed from the bottom up using the precomputed results from the previous step. The following depicts the overall workflow of the improved algorithm:



4.2.1. Criterion [I]: Inclusion

The *inclusion graph* $\mathcal{Q}_{\hookrightarrow}$ is a subgraph of \mathcal{Q} that contains all pairs of simplices that satisfy criterion [I]:

$$\mathcal{Q}_{\hookrightarrow} := (\Sigma_{\geq q}, E_{\hookrightarrow} = \{(\sigma, \tau) \in (\Sigma_{\geq q} \times \Sigma_{\geq q}) \mid \sigma \hookrightarrow \tau\})$$

The computation of E_{\hookrightarrow} forms the first step of the algorithm. By definition, the successors of a simplex in the inclusion graph form the set of simplices that σ is a face of. In turn, the predecessors of a simplex σ are all simplices that are a face of σ , which can be enumerated using the faces operator.

Lemma 4.2.1. *For any simplex $\sigma \in \Sigma_{> q}$, the set of predecessors in the inclusion graph $\delta_{\mathcal{Q}_{\hookrightarrow}}^-(\sigma)$ is equal to $\bigcup_{d=q}^{\dim(\sigma)} \text{faces}(\sigma, d)$.*

Proof. By definition,

$$\delta_{\mathcal{Q}_{\hookrightarrow}}^-(\sigma) = \{\tau \in \Sigma_{\geq q} \mid \tau \hookrightarrow \sigma\}.$$

Because $\tau \hookrightarrow \sigma \Leftrightarrow \tau \in \text{faces}(\sigma, \dim(\tau))$, the predecessors

$$\delta_{\mathcal{Q}_{\hookrightarrow}}^-(\sigma) = \{\tau \in \Sigma_{\geq q} \mid \tau \in \text{faces}(\sigma, \dim(\tau))\} = \bigcup_{d=q}^{\dim(\sigma)} \text{faces}(\sigma, d).$$

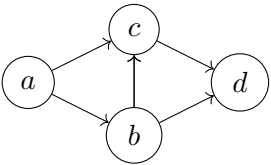
□

Note that any simplex σ is both its own predecessor and successor due to the reflexivity of inclusion. As self loops are omitted for the output graph, $\dim(\sigma) - 1$ will be used as the upper limit of the union for the algorithm.

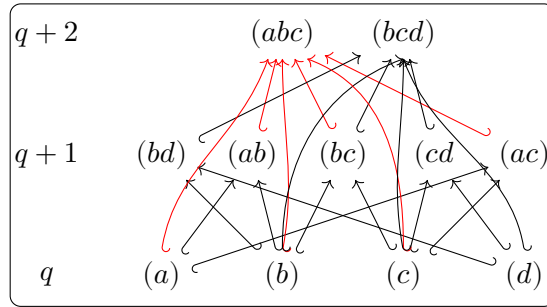
Remark 4.2.2. Similarly for the successors, $\delta_{\mathcal{Q}_{\hookrightarrow}}^+(\sigma) = \bigcup_{d=\dim(\sigma)+1}^{\infty} \{\tau \mid \sigma \in \text{faces}(\tau, \dim(\sigma))\}$ also holds. This is computationally impractical, as it is considerably more difficult to compute the the simplices that contain σ than just enumerating the ordered subsets of σ .

Example 4.2.3. Consider the graph G from Figure 4.3 and its corresponding inclusion graph $\mathcal{Q}_{\hookrightarrow}$. Applying Lemma 4.2.1 to the simplex (abc) yields:

$$\begin{aligned} \delta_{\mathcal{Q}_{\hookrightarrow}}^-((abc)) &= \bigcup_{d=0}^1 \text{faces}((abc), d) \\ &= \text{faces}((abc), 1) \cup \text{faces}((abc), 0) \\ &= \underline{\{(ab), (bc), (ac)\} \cup \{(a), (b), (c)\}} \end{aligned}$$



(a) Example graph G



(b) Inclusion graph $\mathcal{Q}_{\hookrightarrow}$ of G for $q = 0$

Figure 4.3.: Inclusion graph with results of Example 4.2.3 highlighted in red.

Proposition 4.2.4. E_{\hookrightarrow} is the combination of incoming edges $\delta_{Q_{\hookrightarrow}}^-(\sigma)$ for all simplices σ in $\Sigma_{\geq q}$:

$$E_{\hookrightarrow} = \bigcup_{\sigma \in \Sigma_{> q}} \underbrace{\left(\left\{ \bigcup_{d=q}^{\dim(\sigma)} \text{faces}(\sigma, d) \right\} \times \sigma \right)}_{\delta_{Q_{\hookrightarrow}}^-(\sigma)}$$

Proof. We show:

$$E_{\hookrightarrow} \stackrel{(a)}{=} \bigcup_{\sigma \in \Sigma_{\geq q}} (\delta_{Q_{\hookrightarrow}}^-(\sigma) \times \sigma) \stackrel{(b)}{=} \bigcup_{\sigma \in \Sigma_{> q}} (\delta_{Q_{\hookrightarrow}}^-(\sigma) \times \sigma) \stackrel{(c)}{=} \bigcup_{\sigma \in \Sigma_{> q}} \left(\left\{ \bigcup_{d=q}^{\dim(\sigma)} \text{faces}(\sigma, d) \right\} \times \sigma \right)$$

1. (a) is a general property of directed graphs
2. (b) follows from the fact that $\delta_{Q_{\hookrightarrow}}^-(\sigma) = \emptyset$ for $\sigma \in \Sigma_q$
3. (c) follows from Lemma 4.2.1

□

This proposition outlines a procedure to compute E_{\hookrightarrow} , which is formalized in Algorithm 4.

Algorithm 4 Method for computing the inclusion graph.

```

1: function GET_ $E_{\hookrightarrow}$ ( $\Sigma$ )
2:    $E_{\hookrightarrow} \leftarrow \emptyset$ 
3:   for  $p$  from  $q + 1$  to  $D$  do
4:     for  $\sigma \in \Sigma_p$  do
5:       for  $d$  from  $q$  to  $\dim(\sigma) - 1$  do            $\triangleright \dim(\sigma) - 1$  to avoid self-loops
6:         for  $\tau \in \text{faces}(\sigma, d)$  do
7:            $E_{\hookrightarrow}.\text{add}(\tau \hookrightarrow \sigma)$ 
8:         end for
9:       end for
10:    end for
11:  end for
12:  return  $E_{\hookrightarrow}$ 
13: end function

```

Correctness

Lines 3 and 4 of Algorithm 4 iterate through all $\sigma \in \Sigma_{> q}$. Lines 5 and 6 then find the predecessors of each σ according to Lemma 4.2.1. All found edges are then combined into E_{\hookrightarrow} .

Time Complexity

The loop body of Algorithm 4 runs exactly $|E_{\hookrightarrow}|$ times. All loop iterators require constant time per element except for finding $\text{faces}(\sigma, d)$, which is linear in $\dim(\sigma)$. If the size of the simplices is considered a constant, this part of the algorithm runs in $\mathcal{O}(|E_{\hookrightarrow}|)$ time, as one edge is added to E_{\hookrightarrow} in constant time with each iteration and every output edge is enumerated exactly once.

4.2.2. Criterion [II]: $(q + 1)$ -simplices

We recall the property of upward closure from Lemma 3.2.2:

For any two (q, i, j) -near $(q+1)$ -simplices $\mu_\sigma \hookrightarrow \sigma$ and $\mu_\tau \hookrightarrow \tau$, it holds that σ is also (q, i, j) -near to τ .

The property highlights the importance of $(q+1)$ -simplices for the definition of (q, i, j) -nearness: If it is known that two $(q+1)$ -simplices are (q, i, j) -near, then so are their supersimplices. As the supersimplices were already computed in the previous section, the natural next step is to figure out which $(q+1)$ -simplices are (q, i, j) -near. An equivalent definition for (q, i, j) -nearness of $(q+1)$ -simplices using cofaces is useful in this context:

Remark 4.2.5. Two simplices $\mu_\sigma, \mu_\tau \in \Sigma_{q+1}$ are (q, i, j) -near if either of the two conditions is fulfilled:

1. $\mu_\sigma \hookrightarrow \mu_\tau$.
2. There exists a (q) -simplex α such that $\text{coF}_i(\alpha) = \mu_\sigma$ and $\text{coF}_j(\alpha) = \mu_\tau$.

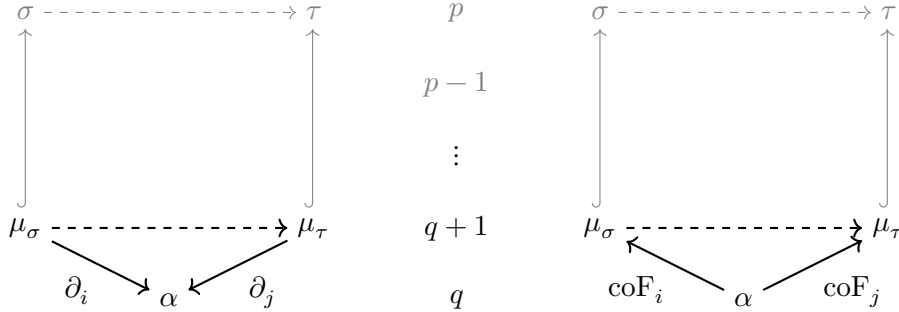


Figure 4.4.: (Left) The original definition of (q, i, j) -near.
(Right) The equivalent alternative definition using the cofaces of (q) -simplices.

The bottom part of Figure 4.4 gives an outline on how to compute all edges of \mathcal{Q} in the dimension $(q+1)$: In the naive algorithm, it was determined for all pairs of simplices in Σ_{q+1} whether their i - and j -faces coincide (as in the left part of Figure 4.4). Using the alternative definition on the right, the problem can be conceptually simplified to computing the i - and j -cofaces for all (q) -simplices: Any i th coface of a (q) -simplex α is

(q, i, j) -near to every j th coface of α . The below equation expresses this formally using E_{q+1} , the set of edges between $(q+1)$ -simplices in the (q, i, j) -near-digraph \mathcal{Q} .

$$E_{q+1} = \bigcup_{\alpha \in \Sigma_q} \left(\text{coF}_i(\alpha) \times \text{coF}_j(\alpha) \right)$$

Algorithm 5 details the necessary computation to find E_{q+1} using cofaces by implementing the above equation. The next section demonstrates how to precompute the required coface datastructures.

Algorithm 5 Compute E_{q+1} using coface maps.

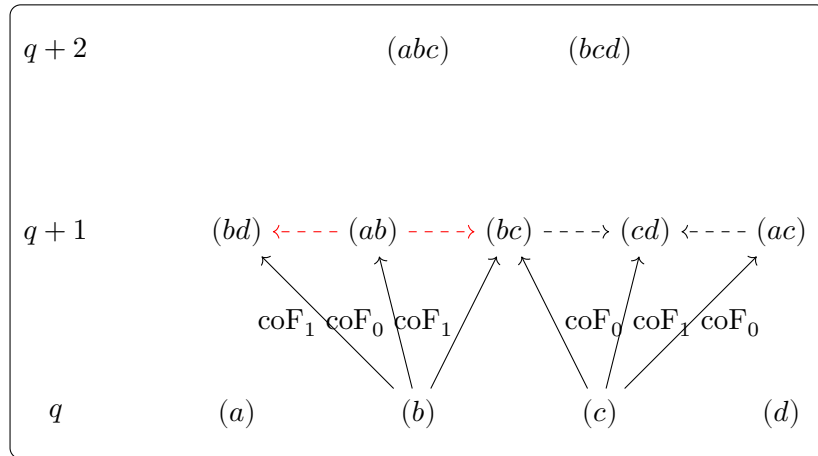
```

1: function GET_ $E_{q+1}(\Sigma_q, \text{coF}_i, \text{coF}_j)$ 
2:    $E_{q+1} \leftarrow \emptyset$ 
3:   for  $\alpha \in \Sigma_q$  do
4:     for  $\sigma \in \text{coF}_i(\alpha)$  do
5:       for  $\tau \in \text{coF}_j(\alpha)$  do
6:          $E_{q+1}.\text{add}(\sigma \rightarrow \tau)$ 
7:       end for
8:     end for
9:   end for
10:  return  $E_{q+1}$ 
11: end function

```

Example 4.2.6. The following Figure depicts the flag complex from Example 4.2.3 with the edges E_{q+1} of the $(0, 0, 1)$ -near-digraph highlighted in red. For this example, we find the edges of \mathcal{Q} between $(q+1)$ -simplices that share the face (b) using the equation above:

$$\begin{aligned} \text{coF}_0(b) \times \text{coF}_1(b) &= \{(ab)\} \times \{(bd), (bc)\} \\ &= \{(ab) \dashrightarrow (bd), (ab) \dashrightarrow (bc)\} \subset E_{q+1} \end{aligned}$$



Time and Space Complexity

Algorithm 5 consists of a straight-forward iteration over the entries of the coface maps. Both coF_i and coF_j contain one entry per $(q + 1)$ -simplex. The time complexity of $\text{GET_}E_{q+1}$ is thus $\mathcal{O}(|\Sigma_{q+1}|)$. The function requires no additional memory aside from the result, which is a subgraph of the output.

Remark 4.2.7. There are two other equivalent implementations of Algorithm 5 that iterate over $(q + 1)$ -simplices σ instead. One of them finds the j -cofaces of $\partial_i(\sigma)$ and the other one the i -cofaces of $\partial_j(\sigma)$. As there are no discernable differences in performance between the versions, the most intuitive approach is presented here.

4.2.3. Criterion [II]: Cofaces

The i -, respectively j -cofaces can be represented as map datastructures coF_i and coF_j from (q) -simplices to sets of $(q + 1)$ simplices. The computation exploits the inverse relation between faces ∂_i and cofaces coF_i detailed in the equations below.

$$\partial_i(\sigma) = \mu_\sigma \Leftrightarrow \sigma \in \text{coF}_i(\mu_\sigma) \text{ and } \text{coF}_i(\mu_\sigma) = \{\sigma \in \Sigma_{\dim(\mu_\sigma)+1} \mid \partial_i(\sigma) = \mu_\sigma\}$$

Directly computing the i th cofaces for all (q) -simplices is expensive, but finding the inverse (i th face of a $(q + 1)$ -simplex) is simple. Every $(q + 1)$ -simplex σ is element of $\text{coF}_i(\mu_\sigma)$ for exactly one $\mu_\sigma \in \Sigma_q$ (and the same holds for j). This decision has already been discussed in the literature ([Lü+20], [UKM23]) and is used in Algorithm 6 to compute coF_i and coF_j efficiently.

Algorithm 6 Construct the coface data structure.

```

1: function GET_SIMPLEX_COFACES( $\Sigma_{q+1}$ )
2:    $\text{coF}_i = \text{new Map } (\Sigma_q \rightarrow \text{List} < \Sigma_{q+1} >)$ 
3:    $\text{coF}_j = \text{new Map } (\Sigma_q \rightarrow \text{List} < \Sigma_{q+1} >)$ 
4:   for  $\sigma \in \Sigma_{q+1}$  do
5:      $\text{coF}_i[\partial_i(\sigma)].\text{append}(\sigma)$ 
6:      $\text{coF}_j[\partial_j(\sigma)].\text{append}(\sigma)$ 
7:   end for
8:   return ( $\text{coF}_i, \text{coF}_j$ )
9: end function

```

Correctness

Lines 5 and 6 of Algorithm 6 exploit the fact that every $(q + 1)$ -simplex σ is both an i -coface of $\partial_i(\sigma)$ and a j -coface of $\partial_j(\sigma)$. The enumeration is complete, because the set of i - and j -cofaces of all (q) -simplices is a subset of Σ_{q+1} .

Time Complexity

For each simplex in Σ_{q+1} , two ∂ operations are performed. These face maps can be computed in constant time (assuming bounded simplex dimension), as this is equivalent to the removal of an array element. As each iteration of the loop requires constant time, the total time complexity of Algorithm 6 is $\mathcal{O}(|\Sigma_{q+1}|)$.

Space Complexity

Precomputing all cofaces as in Algorithm 6 incurs an additional storage overhead of $\mathcal{O}(|\Sigma_{q+1}|)$ (the size of coF_i and coF_j).

4.2.4. Criterion [II]: Higher dimensional simplices

In the previous steps, the edges of \mathcal{Q} that are due to criterion [I] (E_{\hookrightarrow}) and part of the edges from criterion [II] (E_{q+1}) were computed. Let E_{\rightarrow} denote the set of edges in \mathcal{Q} between simplex pairs that satisfy criterion [II]. Then, every edge in E_{\rightarrow} originates from an edge in E_{q+1} , as depicted in Figure 4.5.

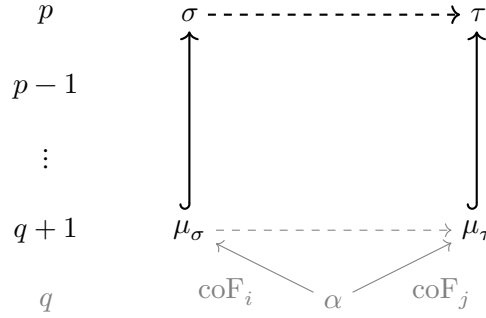


Figure 4.5.: Criterion [II] for higher dimensional simplices.

We now exploit the upward closure property: For every pair $(\mu_s, \mu_\tau) \in E_{q+1}$, all combinations of their supersimplices are also contained in E_{\rightarrow} .

Proposition 4.2.8. *The set of edges in E_{\rightarrow} is equivalent to the combinations of supersimplices for all pairs $(\mu_\sigma \rightarrow \mu_\tau) \in E_{q+1}$.*

$$E_{\rightarrow} = \{\delta_{Q_{\hookrightarrow}}^+(\mu_\sigma) \times \delta_{Q_{\hookrightarrow}}^+(\mu_\tau) \mid (\mu_\sigma, \mu_\tau) \in E_{q+1}\}$$

Proof. The set of successors in the inclusion graph $\delta_{Q_{\hookrightarrow}}^+(\mu_\sigma)$ can also be read as the "set of supersimplices of μ_σ ".

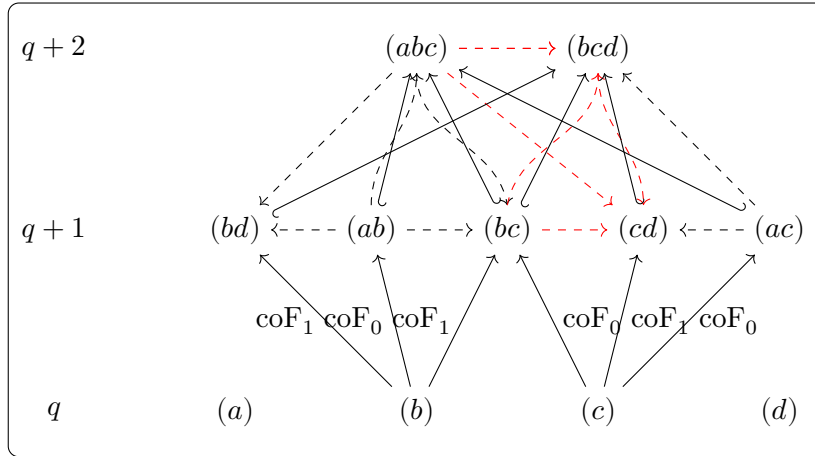
" \supseteq " follows from Lemma 3.2.2 (upward closure): If two $(q+1)$ -simplices are (q, i, j) -near, so are their parents.

" \subseteq " follows from Lemma 3.2.5: If σ, τ are (q, i, j) -near, they must contain $(q + 1)$ -faces that are (q, i, j) -near.

□

Example 4.2.9. Continuing from Example 4.2.6, we explicitly compute the upward closure for the pair $\{(bc), (cd)\}$ using Proposition 4.2.8. The following figure contains all edges of E_{\rightarrow} with the results of the computation highlighted in red.

$$\begin{aligned} \delta_{\mathcal{Q}_{\hookrightarrow}}^+((bc)) \times \delta_{\mathcal{Q}_{\hookrightarrow}}^+((cd)) = \\ \{(abc), (bcd), (bc)\} \times \{(cd), (bcd)\} = \\ \{(abc) \dashrightarrow (cd), (abc) \dashrightarrow (bcd), (bcd) \dashrightarrow (cd), \\ (bcd) \dashrightarrow (bcd), (bc) \dashrightarrow (cd), (bc) \dashrightarrow (bcd)\} \end{aligned}$$



Crucially, both $\mathcal{Q}_{\hookrightarrow}$ and E_{q+1} are already known at this point in the algorithm. The sets $\delta_{\mathcal{Q}_{\hookrightarrow}}^+(\sigma)$ can be precomputed during the inclusion step for all $\sigma \in \Sigma_{q+1}$ with minimal overhead or otherwise found using simple graph search. Algorithm 7 shows an expanded version of Algorithm 5 that uses E_{q+1} to compute the edges in higher dimensions.

Algorithm 7 Compute (q, i, j) -nearness using coface maps and inclusion.

```

1: function GET_ $E_{\rightarrow}$ ( $\Sigma$ ,  $\text{coF}_i$ ,  $\text{coF}_j$ )
2:    $E_{\rightarrow} \leftarrow \text{new HashSet<Edge>}$ 
3:   for  $\alpha \in \Sigma_q$  do
4:     for  $\mu_\sigma \in \text{coF}_i(\alpha)$  do
5:       for  $\mu_\tau \in \text{coF}_j(\alpha)$  do
6:         for  $\sigma \in \delta_{Q_{\hookrightarrow}}^+(\mu_\sigma)$  do
7:           for  $\tau \in \delta_{Q_{\hookrightarrow}}^+(\mu_\tau)$  do
8:              $E_{\rightarrow}.\text{add}(\sigma \rightarrow \tau)$ 
9:           end for
10:        end for
11:      end for
12:    end for
13:  end for
14:  return  $E_{\rightarrow}$ 
15: end function

```

Correctness

The outer part of Algorithm 7 is inherited from Algorithm 5 to enumerate all pairs of (q, i, j) -near simplices in Σ_{q+1} . Then, lines 6 to 8 iterate through all supersimplices of μ_σ and μ_τ to find all edges satisfying criterion [II] according to Proposition 4.2.8. It may be helpful for understanding to compare lines 3 to 5 of Algorithm 7 to Figure 4.4 and lines 6 to 8 to Figure 4.5. The result E_{\rightarrow} uses a HashSet as a datastructure in order to avoid duplication of edges.

Time Complexity

The time complexity per found edge in Algorithm 7 is constant when using the pre-computed coboundaries and inclusion graph. Still, edges may be found multiple times: Simplices σ and τ can share up to $\binom{D}{q+1}$ (q) -faces α . Each of those shared faces α may be contained in D $(q+1)$ -faces μ_σ of σ (and analogously for μ_τ and τ): The face μ_σ can be written as $(\alpha_0 \dots \alpha_{i-1} \sigma_k \alpha_{i+1} \dots \alpha_{q+1})$, where σ_k may be any of the up to D vertices of σ . As such, D^2 is an upper bound for the number of $(q+1)$ -face pairs between σ and τ that share α . Multiplying with the number of possible α yields:

$$\binom{D+1}{q+1} \cdot D^2 \in \mathcal{O}((D+1)^{q+3})$$

While this upper bound is (probably) not tight, it appears that the number of occurrences of each edge in the algorithm scales polynomially in D and exponentially in q . Improving on this upper bound is difficult, as the distribution of simplices over dimensions is highly dependent on the structure of the graph. For practical purposes, D and

q can be considered constants, meaning every edge is found a constant number of times. With this assumption, the time complexity reduces to $\Theta(|E_{\rightarrow}|)$.

4.2.5. Summary

Using the functions described in this section, the edges of \mathcal{Q} that satisfy criterion [I] (using Algorithm 4) and that satisfy criterion [II] (using Algorithm 7) can be calculated. Together, they form the (q, i, j) -digraph \mathcal{Q} . Algorithm 8 shows how to put the parts together to form a finished algorithm computing \mathcal{Q} .

Algorithm 8 The full improved algorithm for computing \mathcal{Q} .

```

1: function GET_ $\mathcal{Q}$ ( $\Sigma$ )
2:    $E_{\hookrightarrow} \leftarrow \text{GET\_}E_{\hookrightarrow}(\Sigma)$   $\triangleright \Theta(|E_{\hookrightarrow}|)$ 
3:    $\text{coF}_i, \text{coF}_j \leftarrow \text{GET\_SIMPLEX\_COFACES}(\Sigma_{q+1})$   $\triangleright \Theta(|\Sigma_{q+1}|)$ 
4:    $E_{\rightarrow} \leftarrow \text{GET\_}E_{\rightarrow}(\Sigma, \text{coF}_i, \text{coF}_j)$   $\triangleright \Theta(|E_{\rightarrow}|)$ 
5:   return ( $\Sigma, E_{\hookrightarrow} \cup E_{\rightarrow}$ )  $\triangleright \Theta(|\mathcal{Q}|)$ 
6: end function

```

Computational Complexity

Assuming the maximal simplex size D is a constant, all steps of Algorithm 8 have a runtime that is linear in the output size $\mathcal{O}(|\mathcal{Q}|)$, because E_{\hookrightarrow} , Σ_{q+1} and E_{\rightarrow} are all subsets of \mathcal{Q} . This would make the time complexity of the algorithm asymptotically optimal. The same holds for the space complexity, as the size of any intermediate results is subsumed by the size of the output \mathcal{Q} .

If D is considered as a variable, the time complexity becomes $\mathcal{O}(D^{q+3}|\mathcal{Q}|)$ due to $\text{GET_}E_{\rightarrow}$. It is unclear at this point if this complexity can be improved.

4.3. An Improved Algorithm for computing $\hat{\mathcal{Q}}$

The algorithm from the previous section may be modified to compute the $(\widehat{q, i, j})$ -near-digraph $\hat{\mathcal{Q}}$ instead. We denote with $\hat{E}_{\hookrightarrow}, \hat{E}_{q+1}$ and \hat{E}_{\rightarrow} the $(\widehat{q, i, j})$ -near analogues of $E_{\hookrightarrow}, E_{q+1}$ and E_{\rightarrow} respectively. Both $\hat{\mathcal{Q}}$ and \mathcal{Q} contain the same inclusion edges ($\hat{E}_{\hookrightarrow} = E_{\hookrightarrow}$) and due to Lemma 3.1.1 also the same edges between $(q+1)$ -simplices ($\hat{E}_{q+1} = E_{q+1}$). Unlike in the previous algorithm, \hat{E}_{\rightarrow} cannot be derived from just $\hat{E}_{\hookrightarrow}$ and \hat{E}_{q+1} , because Lemma 3.2.2 (upward closure) does not hold for $(\widehat{q, i, j})$ -nearness.

4.3.1. Criterion [II]: $(\widehat{q, i, j})$ -near

Simplices σ, τ are $(\widehat{q, i, j})$ -near by criterion [II] if there exists an $\alpha \in \Sigma_q$ such that $\partial_i(\sigma) \leftrightarrow \alpha \hookrightarrow \partial_j(\tau)$. Making use of cofaces, the definition can be rewritten as follows:

Remark 4.3.1. Two simplices $\mu_\sigma, \mu_\tau \in \Sigma_{>q}$ are (q, i, j) -near if either of the two conditions is fulfilled:

1. $\sigma \hookrightarrow \tau$,
2. There exists a q -simplex α and faces μ_σ and μ_τ , such that $\mu_\sigma \hookleftarrow \alpha \hookrightarrow \mu_\tau$ and $\text{coF}_i(\mu_\sigma) = \sigma, \text{coF}_j(\mu_\tau) = \tau$.

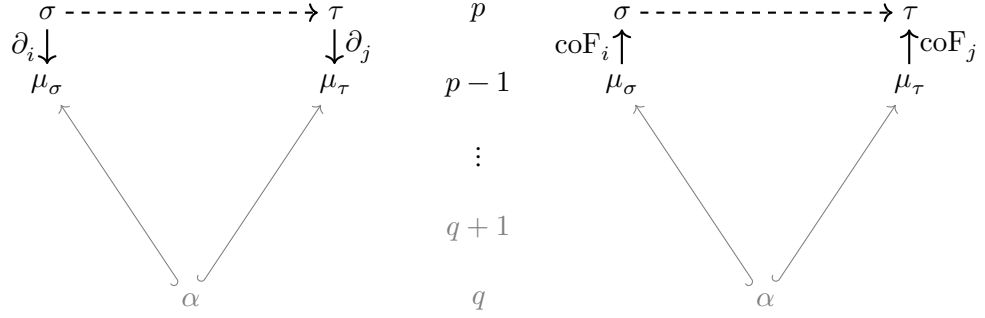


Figure 4.6.: (Left) The original definition of (q, i, j) -near.
(Right) The equivalent alternative definition using cofaces.

For (q, i, j) -nearness, a modified version of Proposition 4.2.8 holds. Here, the order of inclusion and coface are reversed just like they are in Remark 4.3.1 as opposed to Remark 4.2.5:

Proposition 4.3.2. \hat{E}_{\rightarrow} is the Cartesian product of cofaces $\text{coF}_i(\mu_\sigma)$ and $\text{coF}_j(\mu_\tau)$ for all pairs of supersimplices of (q) -simplices α :

$$\hat{E}_{\rightarrow} = \{\text{coF}_i(\mu_\sigma) \times \text{coF}_j(\mu_\tau) \mid (\mu_\sigma, \mu_\tau) \in \{\delta_{\hat{Q}_{\hookrightarrow}}^+(\alpha) \times \delta_{\hat{Q}_{\hookrightarrow}}^+(\alpha)\}, \alpha \in \Sigma_q\}$$

Proof. Assume $(\sigma, \tau) \in \hat{E}_{\rightarrow}$. This implies that

$$\partial_i(\sigma) = \mu_\sigma \in \delta_{\hat{Q}_{\hookrightarrow}}^+(\alpha) \Leftrightarrow \alpha \hookrightarrow \partial_i(\sigma)$$

for some $\alpha \in \Sigma_q$. Similarly, this means

$$\partial_j(\tau) = \mu_\tau \in \delta_{\hat{Q}_{\hookrightarrow}}^+(\alpha) \Leftrightarrow \alpha \hookrightarrow \partial_j(\tau)$$

for the same α . In conclusion,

$$(\sigma, \tau) \in \hat{E}_{\rightarrow} \Leftrightarrow \partial_i(\sigma) \hookleftarrow \alpha \hookrightarrow \partial_j(\tau).$$

□

Using all precomputed cofaces, Algorithm 9 implements Proposition 4.3.2.

Algorithm 9 Compute (q, i, j) -near relations using the inclusion graph

```

1: function GET_ $\hat{E}_{\rightarrow}$ ( $\Sigma$ ,  $\text{coF}_i$ ,  $\text{coF}_j$ )
2:    $E_{\rightarrow} \leftarrow \emptyset$ 
3:   for  $\alpha \in \Sigma_q$  do
4:     for  $\mu_{\sigma} \in \delta_{Q_{\hookrightarrow}}^+(\alpha)$  do
5:       for  $\mu_{\tau} \in \delta_{Q_{\hookrightarrow}}^+(\alpha)$  do
6:         for  $\sigma \in \text{coF}_i(\mu_{\sigma})$  do
7:           for  $\tau \in \text{coF}_j(\mu_{\tau})$  do
8:              $\hat{E}_{\rightarrow}.\text{add}(\sigma \rightarrow \tau)$ 
9:           end for
10:        end for
11:      end for
12:    end for
13:  end for
14:  return  $\hat{E}_{\rightarrow}$ 
15: end function

```

Finding Cofaces

The challenge in converting Proposition 4.3.2 to an algorithm is that the coface operators are now applied to simplices μ_{σ}, μ_{τ} of arbitrary dimension instead of just dimension $(q + 1)$. As such, the i - and j -cofaces of all simplices in $\Sigma_{\geq q}$ have to be precomputed. This can be accomplished by calling $\text{GET_SIMPLEX_COFACES}(\Sigma_{\geq q})$ from Algorithm 6 (note the different inputs).

Time Complexity

Precomputing all cofaces instead of just those from Σ_{q+1} increases the time complexity from $\mathcal{O}(|\Sigma_{q+1}|)$ to $\mathcal{O}(|\Sigma_{\geq q}|)$. For constant D , the time complexity of Algorithm 9 is linear in the output size, analogous to Algorithm 7. A similar upper bound on the number of times each edge may be iterated as for Algorithm 7 also applies: Each edge $(\sigma, \tau) \in \hat{E}_{\rightarrow}$ is iterated once per shared (q) -face α of $\partial_i(\sigma)$ and $\partial_j(\tau)$. The number of possible (q) -faces is bounded by:

$$\binom{D}{q+1} \in \mathcal{O}(D^{q+1}).$$

While this bound improves upon the improved algorithm for (q, i, j) -nearness, in practice this is outweighed by the increased demand of the coface precomputation, as will be shown in Section 5.5.

4.3.2. Summary

Algorithm 10 combines the previously discussed methods into a function that computes the (\widehat{q}, i, j) -near-graph $\hat{\mathcal{Q}}$.

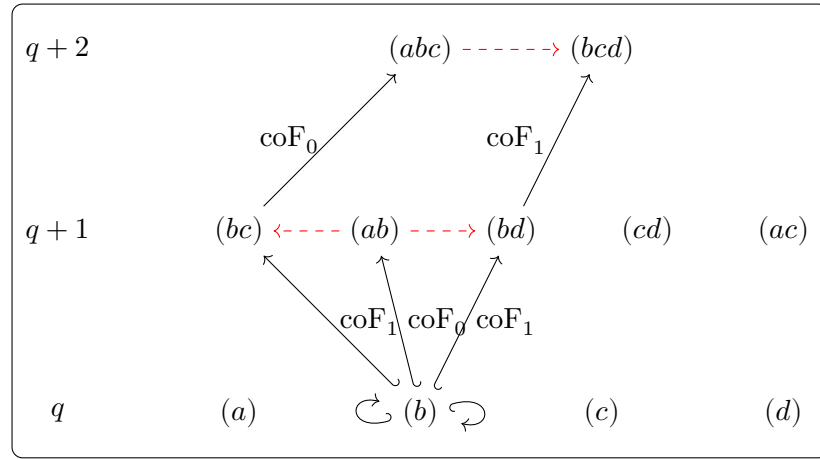
Algorithm 10 Efficient algorithm for computing $\hat{\mathcal{Q}}$

```

1: function GET_ $\hat{\mathcal{Q}}$ ( $\Sigma$ )
2:    $\hat{E}_{\hookrightarrow} \leftarrow \text{GET\_}E_{\hookrightarrow}(\Sigma)$   $\triangleright \Theta(|E_{\hookrightarrow}|)$ 
3:    $\text{coF}_i, \text{coF}_j \leftarrow \text{GET\_SIMPLEX\_COFACES}(\Sigma_{\geq q}, \Sigma_{\geq q})$   $\triangleright \Theta(|\Sigma_{\geq q}|)$ 
4:    $\hat{E}_{\rightarrow} \leftarrow \text{GET\_}\hat{E}_{\rightarrow}(\Sigma)$   $\triangleright \Theta(|\hat{E}_{\rightarrow}|)$ 
5:   return ( $\Sigma, \hat{E}_{\hookrightarrow} \cup \hat{E}_{\rightarrow}$ )  $\triangleright \Theta(|\hat{\mathcal{Q}}|)$ 
6: end function

```

Example 4.3.3. Consider the following flag complex Σ :



In this example, we show that the edges $(ab) \dashrightarrow (bc)$, $(ab) \dashrightarrow (bd)$ and $(abc) \dashrightarrow (bcd)$ are part of \hat{E}_{\rightarrow} for the $(0, 0, 1)$ -digraph using Proposition 4.3.2:

$$\begin{aligned}
(b) \times (b) &\subset \{\delta_{\hat{\mathcal{Q}}_{\hookrightarrow}}^+(b) \times \delta_{\hat{\mathcal{Q}}_{\hookrightarrow}}^+(b)\} \\
\text{coF}_0((b)) \times \text{coF}_1((b)) &= (ab) \dashrightarrow \{(bc), (bd)\} \in \hat{E}_{\rightarrow} \\
(bc) \times (bd) &\subset \{\delta_{\hat{\mathcal{Q}}_{\hookrightarrow}}^+(b) \times \delta_{\hat{\mathcal{Q}}_{\hookrightarrow}}^+(b)\} \\
\text{coF}_0((bc)) \times \text{coF}_1((bd)) &= (abc) \dashrightarrow (bcd) \in \hat{E}_{\rightarrow}
\end{aligned}$$

Computational Complexity

The only difference in computational complexity between the improved Algorithm for the novel and original definition is the increased complexity of precomputing cofaces of all dimensions. For constant maximal simplex dimension D , the time and space complexity for this algorithm is still linear in the output size.

5. Implementation

The algorithms described in Sections 4.1, 4.2, 4.3 and A.1 were implemented in the Rust package `DirQ` (<https://github.com/FelixWindisch/DirQ>). The language was chosen for its low-level control and powerful concurrency features. The computation of the directed flag complexes was built on the Rust implementation of `flagser` [Lü17] found in [UK21] with kind permission from the authors.

5.1. Representing Graphs and Simplices

The input graph G is imported using the `flagser` file format (See [Lü+20] for details) and is stored using an adjacency matrix. The adjacency matrix representation allows for edge lookups in constant time, but has a significant $\mathcal{O}(|V|^2)$ space complexity. In practice, the matrix is stored as a bitfield to ensure that every potential edge only requires a single bit. For the output graph \mathcal{Q} , this method is infeasible, as it contains too many vertices. Instead, the representation as an edge list works well, as \mathcal{Q} is typically sparse. An edge list datastructure can also easily be exported using the `flagser` file format. For both the output and input graph, the vertices are represented using unsigned 64 bit integers (`u64`) and simplices are stored as dynamic arrays (`Vec<u64>`). The entire flag complex can then be stored as `Vec<Vec<Vec<u64>>>` and indexed using dimension, simplex index and vertex index in that order. All simplices of same dimension are stored contiguously in memory, which not only benefits cache locality, but also allows passing Σ_q and Σ_{q+1} as efficient vector slices (`&[Vec<u64>]`) to functions.

5.2. Simplex IDs

Storing copies of simplices is detrimental to both memory usage and runtime. This can be easily avoided by assigning 64 bit IDs to every simplex. The 64 bits are strictly required, as the number of simplices may exceed 2^{32} in practice. A hashmap `Simplex_IDs:HashMap<Vec<u64>, u64>` maps simplices to their ID. Conversion in the other direction is possible (but rarely needed) using a flattened version of the directed flag complex. Edges of \mathcal{Q} are stored as a pair of simplex IDs. The IDs are chosen such that any simplex will have a higher ID than all simplices of lower dimensions. This is useful in multiple contexts:

- It allows passing Σ_q and Σ_{q+1} as vector slices (the Rust equivalent of fat pointers).
- For the computation of \mathcal{Q} , it allows storing coF_i and coF_j as vectors of size $|\Sigma_q|$ where each entry is a vector.

This order means that simplex IDs must be assigned after computation of the directed flag complex, as conventional methods do not enumerate the simplices in order of dimension.

Bottom-Up Algorithm

(see Section A.1) Assigning simplex IDs requires computation and storage of Σ . The bottom-up Algorithm avoids this overhead by always storing simplices as full ordered sets of vertices. This can be seen as a tradeoff between computation time and memory consumption.

5.3. Parallelization

The algorithms are parallelized using the rayon crate [Mat17]. For each of the three major steps of the algorithm, three different methods for parallelization were implemented and tested:

Mutual Exclusion

Each of the steps of the algorithm involves processing simplices and adding elements to a result datastructure. In order, these datastructures are E_{\hookrightarrow} , $\text{coF}_{i/j}$ and E_{\rightarrow} . Therefore, a simple way to parallelize is to split the simplices among different threads to process and then add the results to a shared datastructure in a controlled way. In order to avoid resource conflicts with the shared result, locks are employed. This method is straightforward, but if the number of threads is high, the downtime spent waiting for locks to release is considerable.

Split-and-Merge

The set of simplices is split into smaller chunks and distributed among the threads. Each thread will then append its results to a local datastructure. The results from each chunks are then merged in parallel using a divide-and-conquer strategy. As all intermediary results are thread-local, there is no need for locks, but the overhead for the merging steps is also considerable: Any element in the resulting datastructure of n elements is merged $\log_2(\frac{n}{C}) = \log_2(n) - \log_2(C)$ times, when every thread gets assigned C input simplices. As C is a constant, each element is merged $\mathcal{O}(\log_2(n))$ times. In contrast, a single element (coface or output edge) can be computed in $\mathcal{O}(1)$ (or polynomial in D). This also puts a limit on the possible parallelization, as the effort for merging will exceed the effort of computing the results sequentially after a certain number of threads.

Bottom Up

The extensive need for synchronization in the other approaches stems from the precomputation of the inclusion graph and coboundaries in a top-down manner. In contrast, the parallelization of the Bottom-Up algorithm avoids this bottleneck entirely: Each thread

gets assigned a set of q -simplices and writes the output edges directly to disk. This level of parallelization comes at the cost of many redundant coface computations.

5.4. Using DirQ

There are four possible ways to interact with DirQ:

1. **As a Rust Crate:** DirQ can be used within a Rust project by adding the files from the crate. The following tree shows the layout of modules:

```

directed_q
├── new
│   ├── single_thread
│   ├── mutex
│   ├── split_and_merge
│   ├── bottom_up
│   └── naive
└── original
    ├── single_thread
    ├── mutex
    ├── split_and_merge
    ├── bottom_up
    └── naive

```

There are five different implementation files for both definitions, each one containing a `get_q_digraph` method that will compute a (q, i, j) -digraph and various helpers such as `get_i_j_cofaces`. The `run` method in the `directed_q` module accepts arguments as described in the ReadMe and automatically writes the results to file.

2. **As a Python module:** The Python module “directed_q” can be found on PyPI³ and integrated into any python project. The following shows an example of how to use the Python bindings:

```

import dir_q, networkx as nx, numpy as np
# computes q-digraph from file according to old definition
my_q_graph = nx.DiGraph(dir_q.compute_q_near_graph_new_from_file(
    "test.flag", q=3, i=0, j=4, max_dimension=10))
# computes q-digraph of a random ER-graph according to new definition
my_q_graph2 = nx.DiGraph(dir_q.compute_q_near_graph_old(
    nx.adjacency_matrix(nx.gnp_random_graph(50, 0.3)).toarray()
    .astype(np.uint64), q=3, i=0, j=4, max_dimension=10))

```

³<https://pypi.org/project/directed-q/>

3. **Using the graphical user interface:** Starting the executable without arguments will open a GUI as seen in Figure 5.1. This is the simplest and recommended way to use DirQ.

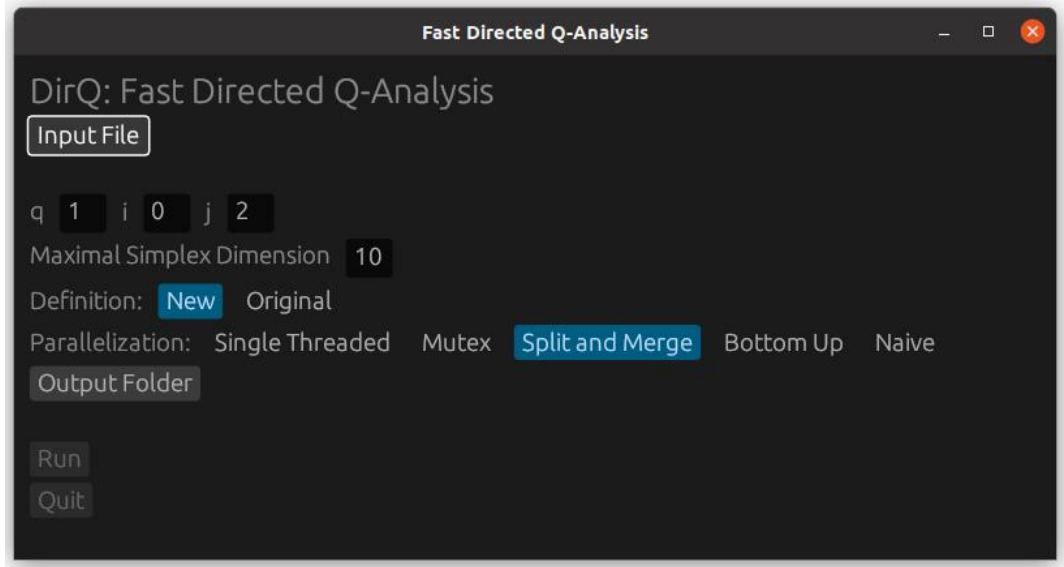


Figure 5.1.: Graphical User Interface of DirQ

4. **Using the command-line interface:** When the executable is started with command line arguments, it will use them instead of starting the GUI. More details for the arguments can be found on [GitHub](#).

5.5. Benchmarks

In order to show the practical efficiency of the presented implementations and algorithms, benchmarks were carried out on various test data. The input graphs include randomly sampled Erdős–Rényi-graphs and neuron-synapse-level connectomes of biological neural networks (The Blue Brain Project’s stochastic reconstruction of a somatosensory cortex of a rodent in *BBP* and the dense reconstruction of the nervous system of the roundworm *C.Elegans*) sourced from [Var+11] and [Coo+19]. All benchmarks were run on a machine with an AMD EPYC 7543 32-Core processor and 512 GB of DDR4 RAM.

Improved Algorithm

Table 5.1 compares the computation time of the optimized Rust implementation of the naive Algorithm 1 (previous state of the art) to the implementation of Algorithm 8 and 10 respectively. The improved algorithm is significantly faster in small testcases and also scales much more efficiently than the naive algorithm. For bigger networks like *BBP*, a 300.000x speedup can be observed. Through this, the improved algorithm enables

previously infeasible analysis of large-scale networks using directed q -nearness. It can also be seen that the novel definition is consistently faster to compute, but only up to a factor of 2. Therefore, both definitions are valid candidates for analysing large-scale networks using the improved algorithm.

Graph	Nodes	Edges	q	Naive		Improved	
				$\widehat{(q, i, j)}$	(q, i, j)	$\widehat{(q, i, j)}$	(q, i, j)
C. Elegans	279	2194	3	10.68s	11.16s	27.93ms	25.72ms
ER-Graph	1000	50k	3	16.98s	17.34s	765.64 μ s	455.97 μ s
BBP	31k	7.6M	4	2062.22s	2054.21s	14.60ms	5.57ms
BBP	31k	7.6M	3	> 48h	> 48h	45.85s	36.18s

Table 5.1.: Benchmark results for the Rust implementation of the naive and efficient algorithm.

Parallelization

Table 5.2 compares the efficiency of Rust implementations for the various discussed parallelization methods. The BBP graph was chosen as a test subject, because it is a practical dataset and large enough to show off the benefits of parallelization. It is immediately apparent that while both mutual exclusion and split-and-merge are improvements on the single-threaded implementation, they do not scale well with the number of cores. For mutual exclusion, the parallelization hits a limit when the waiting list for the mutex is consistently occupied. In fact, one can even see a slight negative effect of using too many cores in Table 5.2, where the increased thread contention leads to decreased performance. The bottom-up algorithm takes a significantly longer time than the improved version, due to its worse asymptotic complexity. On the other hand, the performance scales almost linearly with the number of available cores, realizing parallelization potential that is not available to the improved algorithm. This makes it an ideal candidate for cluster computing, since the individual machines only need a single round of communication at the end.

Graph	q	cores	Single-thread	Mutex	Split-and-Merge	Bottom-Up
BBP	4	2	640ms	540ms	540ms	1201s
BBP	4	8	640ms	490ms	470ms	300s
BBP	4	32	690ms	610ms	450ms	81s

Table 5.2.: Benchmarks for different parallelization methods on the Blue Brain Project graph for different number of cores.

6. Discussion and Outlook

6.1. Summary

Theoretical Perspective

In this work, we introduced the concept of directed q -nearness from a network science perspective and presented a novel competing definition. We showed that the two definitions are identical for some important cases, but not in general: While the original definition describes which part of the simplices intersect, our novel definition focusses on how they intersect. The differences between the definitions was explored using visualizations and combinatorial methods. In particular, we find out that both definitions are equally strict and that the information gain from the directed component diminishes with higher dimension.

Algorithmic Perspective

The original aim of this thesis was to show that the novel definition of directed q -nearness enables significantly faster algorithms. In the end, it turned out that the uncovered algorithmic ideas for (q, i, j) -nearness are also applicable to the original definition. A central pillar of this work is formed by the numerous ways to compute (q, i, j) -digraphs described: The naive algorithm formalizes the design that was used for the analysis in [Rii23]. A combination of top-down and bottom-up approaches is used in the improved algorithm to lower the time complexity, resulting in a speedup of up to 300.000 over the naive algorithm, as shown in benchmarks. A third approach, the pure bottom-up algorithm (see Section A.1), introduces more computational overhead, but lowers the space complexity and features better parallelizability than the improved algorithm. Finally, the matrix-based algorithm (see Section A.2) enables a new, more generalized definition of (q, i, j) -nearness and can be implemented in just a dozen lines of python code.

6.2. Discussion

Comparison of Definitions

The novel definition of directed q -nearness comes with both advantages and drawbacks. The property of upward closure is convenient for algorithms and more complex patterns in the intersection between simplices can be identified. The definition extends to higher-dimensional simplices based on the natural directed nearness between $(q + 1)$ -simplices

and the indices i and j have a well-defined range of $[0, q + 1]$. On the other hand, it is harder to grasp and work with in a lot of cases (such as counting or in the matrix-based algorithm). As shown in the benchmarks, the novel definition is still faster to compute, but both definitions are now feasible for large graphs.

What algorithm to choose?

In the vast majority of cases, the improved algorithm is the most efficient choice for computing (q, i, j) -digraphs. The parallelization using Split-and-Merge can improve the computation time, but also requires some tuning of parameters like chunk size to be effective. In a massively distributed setting, the bottom-up algorithm can better make use of parallelization and requires less local storage, potentially enabling the analysis of networks too massive for single machines.

GPU Acceleration

Implementing algorithms to compute (q, i, j) -digraphs to run on the GPU appears very challenging. The matrix-based algorithm would seem like a prime candidate due to the efficiency of matrix multiplications on graphics cards. The size of the required matrices however renders this approach infeasible on current hardware, as the storage requirements would quickly exceed available VRAM by many factors. The bottom-up algorithm would be a better candidate, as it is massively parallelizable and the input graph required for the computations could easily fit into the shared or global memory of the GPU.

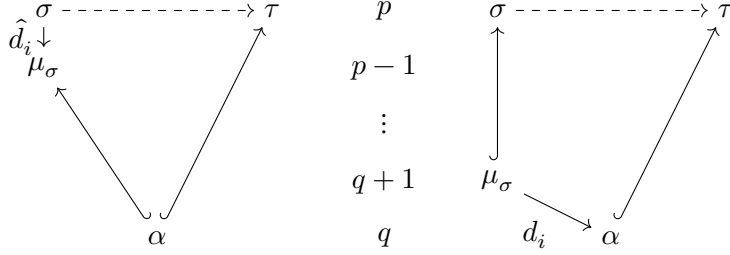
Simplex Dimension Bounds

When using directed q -nearness for analyzing networks, the maximal simplex dimension should be capped: For simplices with significantly more than $q + 1$ vertices, both definitions no longer reflect the directionality of the underlying graph and are almost identical to classic, undirected q -nearness. Additionally, many of the algorithms scale poorly with the dimension of simplices. In any case, networks with few, large simplices are not interesting targets for directed Q -analysis.

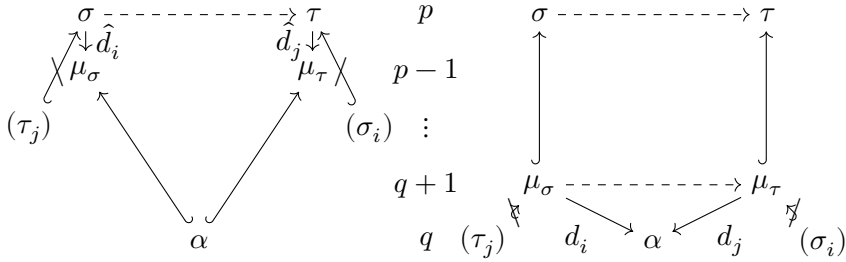
6.3. Outlook

Alternative Definitions

The presented definitions are not the only reasonable way interpret directed q -nearness. Other alternative definitions also warrant further exploration. For example, one may consider a one-sided definition of (q, i) - or $(\widehat{q, i})$ -nearness that only uses one index i :



For both definitions considered in this work, simplices that share a $(q + 2)$ -face or greater are near for all values of i and j , which completely ignores their direction. This would no longer be the case if the definitions were modified to include $\sigma_i \not\leftrightarrow \tau$ and $\tau_j \not\leftrightarrow \sigma$:



The alternative definition that uses weighted vectors i and j enabled by the matrix-based algorithm described in Section A.2 is also an interesting candidate for further study. While the complexity of the algorithm limits its use to smaller networks, the analysis can be done on a much finer scale. The generalized face maps ∂_i can also be used to construct a weighted version of (q, i, j) -nearness, but that definition turns out to be even more computationally intensive and difficult to interpret.

Alternative Complexes

In this work, the directed flag complexes Σ were restricted to those defined by simple graphs. Directed complexes which are not necessarily flag could also be analyzed using (q, i, j) - or $(\widehat{q, i, j})$ -nearness. The main obstacle to overcome is the profound rarity with which directed flag complexes occur outside of graph theory.

Topological Implications

On first glance, q -nearness looks like an alternative to more established methods of topological data analysis like investigating Betti-numbers. However, as Riihimäki pointed out in personal correspondence, Q -analysis could be seen more as a preprocessing step for homology-based topological analysis. In this spirit, it would be highly interesting to experimentally investigate how, if at all, the final results differ with regard to the choice of definition.

Network Analysis

The efficient algorithms presented in this thesis enable analysis of previously out-of-reach networks. This is particularly interesting for computational neuroscience, as there is already established work about analysing neural connectomes using directed simplices ([Rii23], [Rei+17]). In [Rii23], the longest paths in the (q, i, j) -digraphs of various connectomes are analyzed. Preliminary testing has revealed that the properties of the (q, i, j) -digraphs for both definitions are very similar when it comes to analyzing the structure of networks. Some interesting observations that we made on neural connectomes include that the maximal degree of vertices and average path length in the (q, i, j) -digraph is higher when i and j are close to each other. These results will be expanded upon and form a future work.

Bibliography

- [Atk74] R. H. Atkin. “An algebra for patterns on a complex.” In: *International Journal of Man-Machine Studies* 6 (1974), pp. 285–307. ISSN: 0020-7373. DOI: 10.1016/S0020-7373(74)80024-6.
- [Coo+19] Steven J Cook et al. “Whole-animal connectomes of both *Caenorhabditis elegans* sexes”. In: *Nature* 571.7763 (2019), pp. 63–71.
- [Lül17] Daniel Lütgehetmann. *flagser*. <https://github.com/luetge/flagser>. 2017-2021.
- [Lü+20] Daniel Lütgehetmann et al. “Computing persistent homology of directed flag complexes”. In: *Algorithms (Basel)* 13.1 (2020), Paper No. 19, 18. DOI: 10.3390/a13010019.
- [Mat17] Niko Matsakis. *Rayon is a data-parallelism library for Rust*. <https://github.com/rayon-rs/rayon>. 2017.
- [Rei+17] Michael W. Reimann et al. “Cliques of Neurons Bound into Cavities Provide a Missing Link between Structure and Function.” In: *Frontiers Comput. Neurosci.* 11 (2017), p. 48. URL: <http://dblp.uni-trier.de/db/journals/ficn/ficn11.html#ReimannNSTPCDLH17>.
- [Rii23] Henri Riihimäki. “Simplicial q -Connectivity of Directed Graphs with Applications to Network Analysis”. In: *SIAM Journal on Mathematics of Data Science* 5.3 (2023), pp. 800–828. DOI: 10.1137/22M1480021. eprint: <https://doi.org/10.1137/22M1480021>. URL: <https://doi.org/10.1137/22M1480021>.
- [UK21] Florian Unger and Jonathan Krebs. *A computer for the number of almost-d-simplices, building atop flagser*. <https://github.com/flomlo/nads>. 2021.
- [UKM23] Florian Unger, Jonathan Krebs, and Michael G. Müller. “Simplex closing probabilities in directed graphs”. In: *Computational Geometry* 109 (2023), p. 101941. ISSN: 0925-7721. DOI: <https://doi.org/10.1016/j.comgeo.2022.101941>. URL: <https://www.sciencedirect.com/science/article/pii/S0925772122000840>.
- [Var+11] Lav R Varshney et al. “Structural properties of the *Caenorhabditis elegans* neuronal network”. In: *PLoS Comput Biol* 7.2 (2011), e1001066.
- [Wil+23] Virginia Vassilevska Williams et al. *New Bounds for Matrix Multiplication: from Alpha to Omega*. 2023. arXiv: 2307.07970 [cs.DS].

A. Additional Algorithmic Ideas

This section covers algorithmic ideas that are not competitive with the improved algorithms, but give important insight into directed q -nearness and contain possibilities for future development.

A.1. Bottom-Up Algorithm

Both of the improved algorithms to find \mathcal{Q} resp. $\hat{\mathcal{Q}}$ rely on precomputing the flag complex, inclusion graph and certain cofaces. While this leads to a theoretically optimal time complexity, it makes parallelizing the algorithm challenging (See Section 5.3) and may require extensive amounts of RAM to store the precomputed data for large graphs. In this chapter, we develop an algorithm that can compute all these operators on demand and write the results sequentially back to disc, drastically reducing the amount of RAM required at the cost of increasing the computational load.

In order to avoid storing the entire directed flag complex, we will introduce an iterator \mathbf{I}_q that will sequentially compute all (q) -simplices of a given directed graph on demand. The construction of this iterator is not particularly challenging, but outside of the scope of this thesis (see [Lü17] for more details). The basic idea of the algorithm is to start with the (q) -simplices and work upward using coface ($\text{coF}_i(\sigma)$) and supersimplex ($\delta_{Q \hookrightarrow}^+(\sigma)$) computations.

A.1.1. Computing Cofaces

While computing the i th face of a simplex is trivial, finding cofaces is more challenging. Algorithm 11 was adapted from [UKM23] to compute $\text{coF}_i(\sigma)$ for any input simplex σ . It does this by iterating over all vertices v of the graph and deciding whether $(\sigma_0 \dots \sigma_{i-1} v \sigma_i \dots \sigma_n)$ is a directed simplex.

Algorithm 11 Routine to compute i -cofaces of one simplex

```

1: function GET_COFACE( $\sigma \in \Sigma_p, i, G$ )
2:    $\text{coF}_i(\sigma) \leftarrow \emptyset$ 
3:   for  $v \in \left( \bigcap_{0 \leq k < i} \delta_G^-(v_k) \cap \bigcap_{i \leq k \leq p-1} \delta_G^+(v_k) \right)$  do
4:      $\text{coF}_i(\sigma).add(\{\sigma_0, \dots, \sigma_{i-1}\} \cup v \cup \{\sigma_i, \dots, \sigma_{p+1}\})$ 
5:   end for
6:   return  $\text{coF}_i(\sigma)$ 
7: end function

```

Computational Complexity

The intersection operations in Algorithm 11 can be implemented efficiently by binary AND-operations of rows of the adjacency matrix. While this is fast in practice, the asymptotic runtime remains $\mathcal{O}(|V|)$, since all of the vertices of G may need to be iterated. The only memory requirement is $\mathcal{O}(|V| \cdot D)$ for storing up to $|V|$ resulting cofaces.

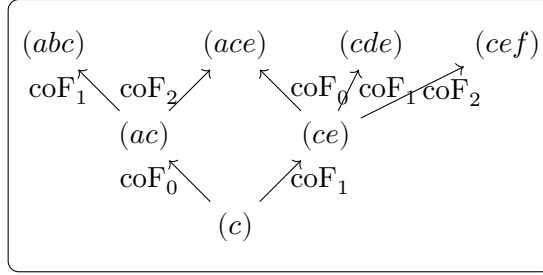


Figure A.1.: Example of a directed acyclic graph induced by cofaces.

A.1.2. Computing Supersimplices

The second required operation is finding the outgoing inclusion edges $\delta_{Q_{\hookrightarrow}}^+(\sigma)$ for some simplex σ . The algorithm can make use of the fact that if $\sigma \hookrightarrow \tau \iff \tau \in \delta_{Q_{\hookrightarrow}}^+(\sigma)$, then there exists some set of indices i_1, \dots, i_k such that $\text{coF}_{i_1} \circ \text{coF}_{i_2} \circ \dots \circ \text{coF}_{i_k}(\sigma) = \tau$. By using Algorithm 11 recursively, all elements of $\delta_{Q_{\hookrightarrow}}^+(\sigma)$ can be enumerated. One may think of a directed acyclic graph (DAG) formed by the cofaces of a simplex and their cofaces and so on. One possible instance of such a *coface DAG* is depicted in Figure A.1. Algorithm 12 computes the set of supersimplices by performing a recursive depth-first search on this DAG.

Algorithm 12 Routine to compute $\delta_{Q_{\hookrightarrow}}^+(\sigma)$ recursively.

```

1: function GET_ $\delta_{Q_{\hookrightarrow}}^+$  ( $\sigma$ , G)
2:   function COMPUTE_ $\delta_{Q_{\hookrightarrow}}^+$ _REC( $\sigma$ , result, G)
3:     for  $i$  in  $[0, |\sigma| + 1]$  do
4:       cofaces  $\leftarrow$  GET_COFA( $\sigma, i, G$ )
5:       for coface in cofaces do
6:         if coface  $\notin$  result then
7:           result.push(coface)
8:           COMPUTE_ $\delta_{Q_{\hookrightarrow}}^+$ _REC(coface, result, G)
9:         end if
10:      end for
11:    end for
12:  end function
13:  supersimplices  $\leftarrow \emptyset$ 
14:  COMPUTE_ $\delta_{Q_{\hookrightarrow}}^+$ _REC( $\sigma$ , supersimplices, G)
15:  return supersimplices
16: end function

```

Computational Complexity

Algorithm 12 implements a depth-first search on the coface DAG. Each step in the DFS is implemented using GET_COFA, which finds at least one edge of the coface DAG in $\mathcal{O}(|V|)$ time complexity. The number of incoming edges for a single simplex is upper-bounded by $(D + 1)$, since every (p) -simplex has exactly $(p + 1)$ faces of dimension $(p - 1)$. In conclusion, for every element of $\delta_{Q_{\hookrightarrow}}^+(\sigma)$, up to $D + 1$ edges of the coface DAG need to be enumerated in $\mathcal{O}(|V|)$ time each, resulting in a total time complexity of $\mathcal{O}(|\delta_{Q_{\hookrightarrow}}^+(\sigma)| \cdot (D + 1) \cdot |V|)$. Besides the result, there is only constant additional storage required.

A.1.3. Computing the Inclusion Graph

Algorithm 12 may be applied to every simplex within $\Sigma_{\geq q}$ in order to compute the inclusion graph Q_{\hookrightarrow} without storing the entire flag complex. This leads to redundant computations however, because if $\sigma \hookrightarrow \tau$, it also holds that $\delta_{Q_{\hookrightarrow}}(\tau) \subset \delta_{Q_{\hookrightarrow}}(\sigma)$. Algorithm 13 shows a modified version of Algorithm 12 that returns all pairs of simplices (σ, τ) where $\sigma \hookrightarrow \tau$ within the explored DAG. By keeping track of the current path during the DFS, inclusion edges are added in each call from all elements in the path to all cofaces of the current simplex. The correctness of these edges follows from the transitivity of inclusion. Because edges may be discovered multiple times in the course of the algorithm, a `HashSet` is used to deduplicate the edges.

Algorithm 13 Routine to compute a subgraph of the inclusion graph for all simplices that contain σ .

```

1: function GET_INCLUSION( $\sigma$ , G)
2:   function COMPUTE_INCLUSION_REC( $\sigma$ , result, path, G)
3:     path.push( $\sigma$ )
4:     for  $i$  in  $[0, |\sigma|]$  do
5:       cofaces  $\leftarrow$  GET_COFA( $\sigma$ ,  $i$ , G)
6:       for coface in cofaces do
7:         for previous in path do
8:           inclusions.insert(previous  $\hookrightarrow$  coface)
9:         end for
10:      GET_INCLUSION_REC(coface, result, path, G)
11:    end for
12:  end for
13: end function
14: inclusions  $\leftarrow$  new HashSet<Edge>
15: path  $\leftarrow \emptyset$ 
16: GET_INCLUSION_REC(coface, inclusion, path, G)
17: return inclusions
18: end function

```

A.1.4. Full Algorithm

Using these building blocks and the (q) -simplex iterator \mathbf{I}_q , a full bottom-up with reduced memory complexity is formalized in Algorithm 14.

Algorithm 14 Bottom-Up algorithm to compute \mathcal{Q} .

```

1: function GET_Q( $G, q, i, j$ )
2:    $E_{\hookrightarrow} \leftarrow$  new HashSet<Edge>
3:   for  $\alpha$  in  $\mathbf{I}_q.get()$  do
4:      $E_{\hookrightarrow} \leftarrow E_{\hookrightarrow} \cup \text{GET\_INCLUSION}(\alpha, G)$ 
5:     for  $\mu_\sigma$  in GET_COFA( $\alpha$ ,  $i$ , G) do
6:       for  $\mu_\tau$  in GET_COFA( $\alpha$ ,  $j$ , G) do
7:         for  $\sigma$  in GET_ $\delta_{\mathcal{Q}_{\hookrightarrow}}(\mu_\sigma, G)$  do
8:           for  $\tau$  in GET_ $\delta_{\mathcal{Q}_{\hookrightarrow}}(\mu_\tau, G)$  do
9:              $E_{\hookrightarrow}.insert(\sigma \rightarrow \tau)$ 
10:          end for
11:        end for
12:      end for
13:    end for
14:  end for
15:  return  $(\Sigma, E_{\hookrightarrow} \cup E_{\hookrightarrow})$ 
16: end function

```

Switching the order of lines 7 – 8 and 9 – 10 results in an equivalent implementation that computes $\hat{\mathcal{Q}}$. The resulting pseudocode can be found in Algorithm 15.

Algorithm 15 Bottom-Up algorithm to compute $\hat{\mathcal{Q}}$.

```

1: function GET_ $\hat{\mathcal{Q}}$ ( $G, q, i, j$ )
2:    $\hat{E}_{\rightarrow} \leftarrow \text{new HashSet}\langle \text{Edge} \rangle$ 
3:   for  $\alpha$  in  $I_q.\text{get}()$  do
4:      $\hat{E}_{\hookrightarrow} \leftarrow \hat{E}_{\hookrightarrow} \cup \text{GET\_INCLUSION}(\alpha, G)$ 
5:     for  $\mu_\sigma$  in  $\text{GET\_}\delta_{\mathcal{Q}_{\hookrightarrow}}^+(\alpha, G)$  do
6:       for  $\mu_\tau$  in  $\text{GET\_}\delta_{\mathcal{Q}_{\hookrightarrow}}^+(\tau, G)$  do
7:         for  $\sigma$  in  $\text{GET\_COF}(\mu_\sigma, i, G)$  do
8:           for  $\tau$  in  $\text{GET\_COF}(\mu_\tau, j, G)$  do
9:              $E_{\rightarrow}.\text{insert}(\sigma \rightarrow \tau)$ 
10:          end for
11:        end for
12:      end for
13:    end for
14:  end for
15:  return  $(\Sigma, E_{\hookrightarrow} \cup E_{\rightarrow})$ 
16: end function

```

Time Complexity

Because Algorithm 14 is in many ways analogous to the improved algorithm, the same upper bound on the number of times that a particular edge $\sigma \dashrightarrow \tau$ can be added to E_{\rightarrow} still applies ($\mathcal{O}(D^{q+3})$). For a given simplex $\sigma \in \Sigma_k$, finding a supersimplex $\sigma' \in \Sigma_{k+1}$ takes $\mathcal{O}(|V|)$ time using the GET_COF routine, which is also used recursively in $\text{GET_}\delta_{\mathcal{Q}_{\hookrightarrow}}^+$. After finding μ_σ and μ_τ in $\mathcal{O}(|V|)$ time, each pair of σ and τ that is found in $\text{GET_}\delta_{\mathcal{Q}_{\hookrightarrow}}^+$ results in at least one edge being added to E_{\rightarrow} . The cost per simplex enumerated by $\text{GET_}\delta_{\mathcal{Q}_{\hookrightarrow}}^+$ is $\mathcal{O}(|V| \cdot (D + 1))$. As edges may be enumerated multiple times, this needs to be applied for every edge up to $\mathcal{O}(D^{q+3})$ times, leading to a total time complexity of $\mathcal{O}(D^{q+4} \cdot |V| \cdot E_{\rightarrow})$ for computing E_{\rightarrow} . For the old definition in Algorithm 15, this is slightly improved to $\mathcal{O}(D^{q+2} \cdot |V| \cdot \hat{E}_{\rightarrow})$, as each edge of \hat{E}_{\rightarrow} may be enumerated only $\mathcal{O}(D^{q+1})$ times.

For the inclusion graph, each call of GET_COF in Algorithm 13 has a time complexity of $\mathcal{O}(|V|)$ and edges for each of the up to D elements in the DFS path have to be generated (in constant time each). At least the edge outgoing from the start of *path* will be added for the first time to E_{\hookrightarrow} . As such, the entire inclusion graph is enumerated in $\mathcal{O}((D + |V|) \cdot E_{\hookrightarrow})$ time. Note that the consideration of D is unnecessary in this case, as $D < |V|$ is guaranteed.

In conclusion, for constant maximal simplex dimension D , the inclusion edges E_{\hookrightarrow} can be computed in $\mathcal{O}(|V| \cdot E_{\hookrightarrow})$ and E_{\rightarrow} in $\mathcal{O}(|V| \cdot E_{\rightarrow})$ time. Compared to the output-sensitive

linear complexity of the improved algorithm, the bottom-up algorithm introduces an additional factor of $|V|$, but skips most of the flag complex precomputation.

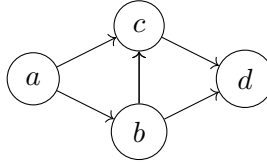
RAM Requirements

The RAM requirements of the bottom-up algorithm can be significantly decreased by writing found edges of \mathcal{Q} directly to external memory. This does not hinder the algorithm, as they do not have to be read afterwards. Because edges may be found multiple times, the result has to be deduplicated after the algorithm terminated. Alternatively, the significantly increased storage of external drives can be used to store the adjacency matrix of the result, rendering deduplication unnecessary. Without the need to store the flag complex, only $\text{coF}(\alpha)$, $\delta_{\mathcal{Q}_{\hookrightarrow}}^+(\mu_\sigma)$ and $\delta_{\mathcal{Q}_{\hookrightarrow}}^+(\mu_\tau)$ for some $\alpha \in \Sigma_q$ and $\mu_\sigma, \mu_\tau \in \Sigma_{>q}$ need to be present in RAM. In extreme cases, $\delta_{\mathcal{Q}_{\hookrightarrow}}^+(\mu_\sigma)$ can contain the entirety of the flag complex. This would only be the case in highly centralized networks, where this type of analysis is not very applicable. In practice, $\delta_{\mathcal{Q}_{\hookrightarrow}}^+(\sigma)$ only represents an insignificant fraction of the entire flag complex for any given $\sigma \in \Sigma_{\geq q}$. Comparing the RAM requirements to the improved algorithm, which needs to keep the entire inclusion graph and flag complex loaded, this represents a notable advantage.

A.2. Matrix-based Algorithm

Finding the size of the largest shared face of two simplices is a common objective in Q-analysis. Solving this problem via set operations is neither elegant nor efficient, as can be seen in Algorithms 2 and 3. However, when the simplices are encoded as binary vectors with as many elements as nodes in the graph, such that ones indicate that the vertex is part of the simplex, their dot product will be equal to the size of the largest shared subsimplex.

Example A.2.1. Consider the simplices $\sigma = (abc)$ and $\tau = (bcd)$ in the following graph:



The respective binary encodings of σ and τ are then

$$\bar{\sigma} = \begin{matrix} a: \\ b: \\ c: \\ d: \end{matrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad \text{and} \quad \bar{\tau} = \begin{matrix} a: \\ b: \\ c: \\ d: \end{matrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \end{bmatrix}.$$

The result of the dot product $\bar{\sigma} \cdot \bar{\tau} = 2$ is then the number of shared vertices between σ and τ .

The encodings of all simplices can be combined into the *incidence matrix*, which allows us to represent entire steps of the algorithm as a single matrix multiplication. However, linear algebra alone is not expressive enough to calculate the directed (q, i, j) -digraph. A few non-linear operators are required for the algorithm to work.

Definition A.2.2. Let $G = (V, E)$ be a simple, directed graph and Σ the corresponding flag complex.

- For a matrix $\mathbf{M} \in \mathbb{N}^{n \times m}$, let \mathbf{M}_{kl} denote the element in the k th row and l th column.
- The incidence matrix $\mathbf{B}^\Sigma \in \mathbb{N}^{|\Sigma| \times |V|}$ is defined as follows:

$$\mathbf{B}_{kl}^\Sigma = \begin{cases} 1 & \text{if } v_l \in \sigma_k, \\ 0 & \text{else.} \end{cases}$$

- Face maps $\partial_i(\mathbf{B}^\Sigma) \in \mathbb{N}^{|\Sigma| \times |V|}$ are defined on an incidence matrix in the following way:

$$\partial_i(\mathbf{B}^\Sigma)_{kl} = \begin{cases} 1 & \text{if } l \in \partial_i(\sigma_k), \\ 0 & \text{else.} \end{cases}$$

This is defined analogously for $\hat{\partial}_i(B^\Sigma)$.

- Let $(\mathbf{M})_{>p}$ denote the thresholded matrix with threshold p :

$$\left((\mathbf{M})_{>p}\right)_{kl} = \begin{cases} 1 & \text{if } \mathbf{M}_{kl} > p, \\ 0 & \text{else.} \end{cases}$$

- The argmax operator inserts a 1 into the position of the maximal element in each row:

$$\text{argmax}(\mathbf{M})_{kl} = \begin{cases} 1 & \text{if } \forall l' \in 0 \dots n : \mathbf{M}_{kl'} \leq \mathbf{M}_{kl}, \\ 0 & \text{else.} \end{cases}$$

The product of the incidence matrix with itself results in a symmetric matrix where any (i, j) -entry corresponds to the size of the largest shared face between the i th and j th simplex. When the shared face is the size of the i th or j th simplex, then it must be a face of the other. This enables the computation of the inclusion graph using matrix multiplication.

Lemma A.2.3. Let Σ be the flag complex of a simple, directed graph G and $\mathbf{Q}_{\hookrightarrow}$ the adjacency matrix representation of the inclusion graph $\mathcal{Q}_{\hookrightarrow}$. Then

$$\mathbf{Q}_{\hookrightarrow} = \text{argmax}(\mathbf{B}^\Sigma \times \mathbf{B}^{\Sigma T}).$$

Proof. Let $\bar{\sigma}_i$ denote the binary encoding of the i th simplex of Σ as described. Then,

$$(\mathbf{B}^\Sigma \times \mathbf{B}^{\Sigma^T})_{kl} = \bar{\sigma}_k \cdot \bar{\sigma}_l$$

For the diagonal elements, the entries correspond to the number of vertices of the simplex: $(\mathbf{B}^\Sigma \times \mathbf{B}^{\Sigma^T})_{kk} = \bar{\sigma}_k \cdot \bar{\sigma}_k = |\sigma_k|$. The diagonal element also represents an upper bound on the elements in the respective row and column. This is because the dot product of binary vectors is akin to a bitwise AND operation, which implies $\bar{\sigma}_l \cdot \bar{\sigma}_k \leq \min\{|\sigma_l|, |\sigma_k|\}$. The diagonal element will always be a maximum of each row, from which follows $\forall k : (\mathbf{Q}_{\hookrightarrow})_{kk} = 1$. This matches the fact that the inclusion relation is reflexive. For any $\sigma_k \hookrightarrow \sigma_l$, the product $\bar{\sigma}_k \cdot \bar{\sigma}_l = |\sigma_k|$ is equal to the diagonal element of the row, so it must be a maximum as well and thus $(\mathbf{Q}_{\hookrightarrow})_{kl} = 1$. If $\sigma_k \not\hookrightarrow \sigma_l$, there must be at least one vertex $v \in \sigma_k$ such that $v \notin \sigma_l$. Thus $\bar{\sigma}_k \cdot \bar{\sigma}_l \leq |\sigma_k| - 1$, which is not a row maximum. In conclusion, $(\mathbf{Q}_{\hookrightarrow})_{kl} = 1$ iff $\sigma_k \hookrightarrow \sigma_l$. \square

Two simplices are $(\widehat{q, i, j})$ -near whenever the i th face of one shares a q -face with the j th face of the other. The product of the incidence matrices of i th faces and j th faces has the size of the largest shared faces as entries. When an entry is greater than q , the two simplices are $(\widehat{q, i, j})$ -near:

Lemma A.2.4. *Let Σ be the flag complex of a simple, directed graph G and $\hat{\mathbf{Q}}_{\rightarrow}$ the adjacency matrix representation of $\hat{\mathcal{Q}}_{\rightarrow}$. Then,*

$$\hat{\mathbf{Q}}_{\rightarrow} = \left(\hat{\partial}_i(\mathbf{B}^\Sigma) \times \hat{\partial}_j(\mathbf{B}^\Sigma)^T \right)_{>q}$$

Proof. In the resulting matrix, an entry $(\hat{\mathbf{Q}}_{\rightarrow})_{kl}$ is equal to 1 iff the dot product of the k th row of $\hat{\partial}_i(\mathbf{B}^\Sigma)$ and the l th row of $\hat{\partial}_j(\mathbf{B}^\Sigma)$ is greater than q . Note that the k th row of $\hat{\partial}_i(\mathbf{B}^\Sigma)$ contains the binary vector encoding of $\hat{\partial}_i(\sigma_k)$, where σ_k is the k th simplex of Σ . Analogously, the l th row of $\hat{\partial}_j(\mathbf{B}^\Sigma)$ contains the binary vector encoding of $\hat{\partial}_j(\sigma_l)$. As already shown, the dot product between the binary representations of the simplices yields the size of the largest shared simplex. It follows that $(\hat{\mathbf{Q}}_{\rightarrow})_{kl} = 1$ iff only $\hat{\partial}_i(\sigma_k)$ and $\hat{\partial}_j(\sigma_l)$ share a (q) -face, which matches the definition of $(\widehat{q, i, j})$ -nearness. \square

Finding a closed-form solution for (q, i, j) -nearness turns out to be significantly more challenging. With matrix multiplication it is easy to answer queries of the form $\exists \alpha \in \Sigma_q : \sigma \hookleftarrow \alpha \hookrightarrow \tau$, but this structure is not directly contained in the new definition.

Proposition A.2.5. *Let Σ be the flag complex of a simple, directed graph G and \mathbf{Q}_{\rightarrow} the adjacency matrix representation of $\mathcal{Q}_{\rightarrow}$. Let $\mathbf{B}_{q+1}^\Sigma \in \mathbb{N}^{|\Sigma_{q+1}| \times |V|}$ be the matrix containing all rows of \mathbf{B}^Σ that sum up to $(q+1)$ and $\mathbf{C}^\Sigma = (\mathbf{B}_{q+1}^\Sigma \times \mathbf{B}^{\Sigma^T})_{>q+1}$. Then,*

$$\mathbf{Q}_{\rightarrow} = \left(\mathbf{C}^{\Sigma^T} \times \left(\partial_i(\mathbf{B}_{q+1}^\Sigma) \times \partial_j(\mathbf{B}_{q+1}^\Sigma)^T \right)_{>q} \times \mathbf{C}^\Sigma \right)_{>0}$$

The idea is very similar to Algorithm 8. The middle section results in a $|\Sigma_{q+1}| \times |\Sigma_{q+1}|$ matrix that encodes E_{q+1} (compare to Equation A.2.4). The matrix \mathbf{C}^Σ encodes inclusion of $(q+1)$ -simplices in higher dimensional simplices. As such, the equation can be read as: A simplex σ is (q, i, j) -near to τ iff it contains a $(q+1)$ -simplex (left part) that is (q, i, j) -near to another $(q+1)$ -simplex (middle part) that is contained in τ (right part).

Proposition A.2.6. *Let Σ be the flag complex of a simple, directed graph G . Let \mathbf{Q} be the adjacency matrix of \mathcal{Q} and $\hat{\mathbf{Q}}$ the adjacency matrix of $\hat{\mathcal{Q}}$ and $\mathbf{Q}_{\rightarrow}, \mathbf{Q}_{\leftrightarrow}, \hat{\mathbf{Q}}_{\rightarrow}$ be defined as in above lemmata.*

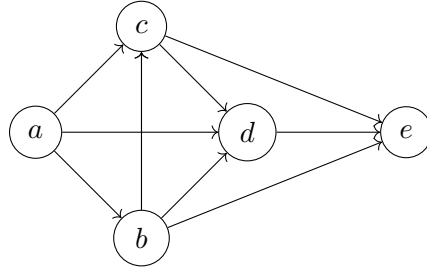
$$\mathbf{Q} = \left(\mathbf{Q}_{\rightarrow} + \mathbf{Q}_{\leftrightarrow} \right)_{>0}$$

$$\hat{\mathbf{Q}} = \left(\hat{\mathbf{Q}}_{\rightarrow} + \mathbf{Q}_{\leftrightarrow} \right)_{>0}$$

Proof. This is analogous to line 5 of Algorithm 8, just in adjacency matrix representation. The threshold $_{>0}$ prevents numbers greater than 1 in the output, as some edges may be found multiple times. \square

A.2.1. Example

To demonstrate the matrix algorithm, we show how to calculate the $(\widehat{2, 0, 2})$ -near-digraph $\hat{\mathcal{Q}}$ of the following graph G :



The first step is to encode the directed flag complex Σ into the incidence matrix \mathbf{B}^Σ :

$$\mathbf{B}^\Sigma = \begin{matrix} & \begin{matrix} a & b & c & d & e \end{matrix} \\ \begin{matrix} (abc) \\ (bcd) \\ (acd) \\ (abd) \\ (bce) \\ (bde) \\ (cde) \\ (abcd) \\ (bcde) \end{matrix} & \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{bmatrix} \end{matrix}$$

Then, the adjacency matrix of the inclusion graph is calculated using Lemma A.2.3 (the bold elements indicate a row maximum):

$$\mathbf{B}^\Sigma \times \mathbf{B}^{\Sigma^T} = \begin{array}{c} \begin{matrix} & (abc) & (bcd) & (acd) & (abd) & (bce) & (bde) & (cde) & (abcd) & (bcde) \end{matrix} \\ \begin{matrix} (abc) \\ (bcd) \\ (acd) \\ (abd) \\ (bce) \\ (bde) \\ (cde) \\ (abcd) \\ (bcde) \end{matrix} \end{array} \begin{bmatrix} \mathbf{3} & 2 & 2 & 2 & 2 & 1 & 1 & \mathbf{3} & 2 \\ 2 & \mathbf{3} & 2 & 2 & 2 & 2 & 2 & \mathbf{3} & \mathbf{3} \\ 2 & 2 & \mathbf{3} & 2 & 1 & 1 & 2 & \mathbf{3} & 2 \\ 2 & 2 & 2 & \mathbf{3} & 1 & 2 & 1 & \mathbf{3} & 2 \\ 2 & 2 & 1 & 1 & \mathbf{3} & 2 & 2 & 2 & \mathbf{3} \\ 1 & 2 & 1 & 2 & 2 & \mathbf{3} & 2 & 1 & \mathbf{3} \\ 1 & 2 & 2 & 1 & 2 & 2 & \mathbf{3} & 2 & \mathbf{3} \\ \mathbf{3} & \mathbf{3} & \mathbf{3} & \mathbf{3} & 2 & 1 & 2 & \mathbf{4} & 3 \\ 2 & 3 & 2 & 2 & 3 & 3 & 3 & 3 & \mathbf{4} \end{bmatrix}$$

Using Equation A.2.4, we can find all $(\widehat{2,0,2})$ -near simplices that do not include one another (bold elements are greater than q):

$$(\partial_0(\mathbf{B}^\Sigma) \times \partial_2(\mathbf{B}^\Sigma)^T) = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$= \begin{array}{c} \begin{matrix} & (abc) & (bcd) & (acd) & (abd) & (bce) & (bde) & (cde) & (abcd) & (bcde) \end{matrix} \\ \begin{matrix} (abc) \\ (bcd) \\ (acd) \\ (abd) \\ (bce) \\ (bde) \\ (cde) \\ (abcd) \\ (bcde) \end{matrix} \end{array} \begin{bmatrix} 1 & 2 & 1 & 1 & 2 & 1 & 1 & 2 & 2 \\ 0 & 1 & 1 & 0 & 1 & 1 & 2 & 1 & 2 \\ 0 & 1 & 1 & 0 & 1 & 1 & 2 & 1 & 2 \\ 1 & 1 & 0 & 1 & 1 & 2 & 1 & 1 & 2 \\ 0 & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 2 & 1 & 1 & 2 & 2 & 2 & 2 & \mathbf{3} \\ 0 & 1 & 1 & 0 & 1 & 1 & 2 & 1 & 2 \end{bmatrix}$$

Finally, we add $\mathbf{Q}_{\hookrightarrow}$ and $\hat{\mathbf{Q}}_{\rightarrow}$ according to Proposition A.2.6 to find the adjacency matrix

of \hat{Q} :

$$\hat{Q} = \begin{matrix} & \begin{matrix} (abc) & (bcd) & (acd) & (abd) & (bce) & (bde) & (cde) & (abcd) & (bcde) \end{matrix} \\ \begin{matrix} (abc) \\ (bcd) \\ (acd) \\ (abd) \\ (bce) \\ (bde) \\ (cde) \\ (abcd) \\ (bcde) \end{matrix} & \begin{bmatrix} \mathbf{1} & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & \mathbf{1} & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ 0 & 0 & \mathbf{1} & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} & \mathbf{1} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \end{bmatrix} \end{matrix}$$

A.2.2. Limitations and Drawbacks

The matrix multiplication algorithm for directed q -nearness only works under the assumption that the original graph G is oriented. Because the order of the shared face is not taken into account, a simplex (abc) would falsely be considered $(1, \widehat{0, \infty})$ -near to (cba) , because it would not distinguish between the face (bc) and (cb) (their binary encodings would be identical). Such a case could not happen in an oriented graph, as the order of simplices would be unique.

Time and Space Complexity

The practical complexity of matrix multiplication is an active area of research and regularly revised [Wil+23]. Even assuming an optimistic $\mathcal{O}(n^{2.4})$ complexity for multiplication, the runtime is significantly increased over the previously presented algorithms, since the size of the matrices is $n = |\Sigma_{\geq q}|$. Results from binary matrix multiplication are not applicable in this case, as the result from the multiplications is not binary. The size of the matrices also leads to a space complexity of $\mathcal{O}(|\Sigma_{\geq q}|^2)$. Sparse matrix data-structures alleviate the storage problems, but the computation time remains an issue, as face maps and threshold operations alone have a total time complexity of $\mathcal{O}(|\Sigma_{\geq q}|^2)$.

A.2.3. Advantages

The matrix-based algorithm is the most compact of all the presented algorithms and the simplest to implement assuming access to linear algebra libraries. The formula to compute \hat{Q} gives a new perspective on the nature of (q, i, j) -nearness. This method also allows rapid testing of different alternative definitions for directed q -nearness by making small changes to the matrix equations. One such alternative definition that is enabled by the matrix-based algorithm is discussed in the following section.

Weighted (q, i, j) -nearness

As discussed in Section 3, many different simplex pairs may be considered $(\widehat{q, i, j})$ -near for the same values of (q, i, j) , particularly for simplices of higher dimension. For example, the simplices from the arrow diagrams in Figure A.2 are both $(\widehat{3, 0, \infty})$ -near:

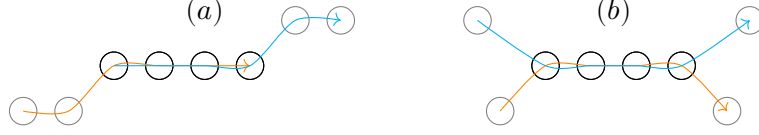


Figure A.2.: Arrow diagrams of simplex pairs that are both $(\widehat{3, 0, \infty})$ -near.

The first simplex pair exhibits the flow we have come to expect from the $i = 0, j = \infty$ direction, but the directionality of the second pair appears different, even though they are also $(\widehat{3, 0, \infty})$ -near. One way to distinguish such cases is by redefining the meaning of the face (∂_i) operator. From the perspective of binary representations of simplices, ∂_i sets the i th 1-entry of the vector to 0. But what if instead every 1-element was replaced by a particular weight? Instead of indices, the parameters i and j could be interpreted as vectors of weights for each vertex:

Definition A.2.7 (∂_i) . Let $\bar{\sigma}$ be the binary encoding of some simplex σ and $\mathbf{i} \in \mathbb{R}^n$. Then,

$$(\partial_{\mathbf{i}}(\bar{\sigma}))_k = \begin{cases} \mathbf{i}_l, l = \sum_{j=0}^{k-1} \bar{\sigma}_j & \text{if } \bar{\sigma}_k = 1, \\ 0 & \text{else.} \end{cases}$$

This is a generalization of the ∂ operator that allows weights instead of just binary vectors. For example, $\partial_{\mathbf{i}}(\sigma) = \partial_0(\sigma)$ for $\mathbf{i} = [0 \ 1 \ 1 \ 1 \dots]^T$.

Definition A.2.8 $(\widehat{(q, \mathbf{i}, \mathbf{j})})$ -nearness). Two simplices σ, τ are $(\widehat{(q, \mathbf{i}, \mathbf{j})})$ -near with $\mathbf{i}, \mathbf{j} \in \mathbb{R}^n$ if for their binary vector representations $\bar{\sigma}$ resp. $\bar{\tau}$

$$(\partial_{\mathbf{i}}(\bar{\sigma})) \cdot (\partial_{\mathbf{j}}(\bar{\tau})) \geq (q + 1)$$

holds, where \cdot denotes the vector dot product.

This definition generalizes $(\widehat{q, i, j})$ -nearness and allows for more fine-grained analysis.

Example A.2.9. Consider the simplex pairs (a) and (b) from Figure A.2. For values $\mathbf{i} = [0 \ 1 \ 1 \ 1 \ 1 \ 1 \dots]^T$ and $\mathbf{j} = [1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1 \dots]^T$, $(\widehat{(q, \mathbf{i}, \mathbf{j})})$ -nearness is equivalent to $(\widehat{q, 0, 6})$ -nearness. The following calculation shows that both simplex pairs are $(\widehat{3, 0, 6})$ -near (vertex indices are in order from left to right in the drawing):

$$\begin{aligned} \partial_{\mathbf{i}}([1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0]^T) &= [0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0]^T \\ \partial_{\mathbf{j}}([0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1]^T) &= [0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0]^T \\ (\partial_{\mathbf{i}}(\bar{\sigma}_1)) \cdot (\partial_{\mathbf{j}}(\bar{\tau}_1)) &= 4 \geq q + 1 \end{aligned} \tag{a}$$

$$\begin{aligned}
\partial_{\mathbf{i}}([1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1]^T) &= [0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1]^T \\
\partial_{\mathbf{j}}([0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0]^T) &= [0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0]^T \\
(\partial_{\mathbf{i}}(\bar{\sigma}_2)) \cdot (\partial_{\mathbf{j}}(\bar{\tau}_2)) &= 4 \geq q + 1.
\end{aligned} \tag{b}$$

For values $\mathbf{i} = [0 \ 0 \ 0.5 \ 1 \ 1.5 \ 2]^T$ and $\mathbf{j} = [2 \ 1.5 \ 1 \ 0.5 \ 0 \ 0]^T$, only the first pair is $(\widehat{3, \mathbf{i}, \mathbf{j}})$ -near:

$$\begin{aligned}
(\partial_{\mathbf{i}}([1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0 \ 0]^T)) &= [0 \ 0 \ 0.5 \ 1 \ 1.5 \ 2 \ 0 \ 0]^T \\
(\partial_{\mathbf{j}}([0 \ 0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1]^T)) &= [0 \ 0 \ 2 \ 1.5 \ 1 \ 0.5 \ 0 \ 0]^T \\
(\partial_{\mathbf{i}}(\sigma)) \cdot (\partial_{\mathbf{j}}(\tau)) &= 5 \geq q + 1
\end{aligned} \tag{a}$$

$$\begin{aligned}
(\partial_{\mathbf{i}}([0 \ 1 \ 1 \ 1 \ 1 \ 1 \ 1 \ 0]^T)) &= [0 \ 0 \ 0 \ 0.5 \ 1 \ 1.5 \ 2 \ 0]^T \\
(\partial_{\mathbf{j}}([1 \ 0 \ 1 \ 1 \ 1 \ 1 \ 0 \ 1]^T)) &= [2 \ 0 \ 1.5 \ 1 \ 0.5 \ 0 \ 0 \ 0]^T \\
(\partial_{\mathbf{i}}(\sigma)) \cdot (\partial_{\mathbf{j}}(\tau)) &= 1 < q + 1
\end{aligned} \tag{b}$$

The example demonstrates how the weighted definition can be used to distinguish the directionality of simplex pairs that are equivalent for $(\widehat{q, i, j})$ -nearness. While this definition is very powerful, it also comes at great computational cost, as it can only be computed using the matrix-based or naive algorithm. As such, this type of analysis is currently limited to small networks. An implementation of the matrix-based algorithm for (q, i, j) - and $(\widehat{q, i, j})$ -digraphs using numpy is included in ⁴.

⁴https://github.com/FelixWindisch/DirQ/blob/main/supplementals/matrix_based.py