Using Graph Traversal in Scientific Data Interpolation

Alireza Rezaei Mahdiraji
Jacobs-University
Bremen, Germany
a.rezaeim@jacobs-university.de

Peter Baumann
Jacobs-University
Bremen, Germany
pbaumann@jacobs-university.de

ABSTRACT

In this paper, we present a topological neighborhood expression which allows us to express arbitrary neighborhood around cells in unstructured meshes. We show that the expression can be evaluated by traversing the connectivity information of the meshes. We implemented two algebraic operators which use the expression to compute neighbors of cells and approximate data fields of cells by aggregating their neighbors' information. We evaluate one of the operators on a real dataset using four queries and report the results.

Keywords

Graph Data Model, Halo, Hull, Regrid, Topological Neighborhood, Unstructured Meshes

1. INTRODUCTION

Many scientific domains such as oceanography and climatology have data stored on unstructured meshes. Weighted contribution from nearest-neighbor cells is known to improve accuracy of interpolation operations on unstructured meshes. Examples of such operations are smoothing a skewed data field, and computing partial derivative in a point of interest.

The common method to specify a neighborhood for a cell of interest is *stencil string* which is originally defined only for structured meshes. Stencil allows us to define the value of a cell as a function of its topological nearest-neighbor cells. In [3], the concept of stencil is generalized for unstructured meshes. A stencil string w.r.t. an unstructured mesh consists of a sequence of digits representing the dimensions of cells in the neighborhood of a cell of interest which needs to be accessed by an algorithm. The stencil string uses hard coded dimensions and thus contains no topological abstraction. Furthermore, it is not obvious from the string what is the result, i.e., union of elements visited in each intermediate layer (*hull*) or the elements only in the last layer (*halo*). Finally, it is not possible to filter intermediate cells using predicates.

©2015, Copyright is with the authors. Published in the Workshop Proceedings of the EDBT/ICDT 2015 Joint Conference (March 27, 2015, Brussels, Belgium) on CEUR-WS.org (ISSN 1613-0073). Distribution of this paper is permitted under the terms of the Creative Commons license CC-by-nc-nd 4.0.

In this paper, we present a neighborhood expression which uses topological functions instead of dimensions and allows filtering of intermediate results. We design two algebraic operators which use the expression to extract neighborhood and approximate information of a cell by interpolating information of its neighbors.

The paper is organized as follows: Section 2 introduces some mathematical concepts, Section 3 discusses the related work, Section 4 presents the new neighborhood expression, Section 5 shows the algebraic operators which use the expression to explore the neighborhood and interpolate data, Section 6 reports the experiments, and Section 7 concludes the paper.

2. MATHEMATICAL CONCEPTS

We define a mesh as a 4-tuple $M = (\mathcal{C}, \prec_{\mathcal{C}}, \Gamma, \mathcal{F})$ where $\mathcal{C}$, $\prec_{\mathcal{C}}$, Γ, and $\mathcal{F}$ are the set of cells, incidences, geometric embedding, and data fields of the mesh M.

$\mathcal{C}$ contains a set of k-cells ($0 \leq k \leq d$) which is closed under intersection, i.e., the intersection of two cells is either empty or another lower dimension cell of the mesh. Each cell in turn is formed by the union of lower dimensional cells (aka the cell boundary). Cells of dimension i are called i-cells. Cells of dimensions zero, one, two, and three are also called vertices, edges, faces, and bodies. Dimension of a mesh is defined as the maximal dimension of its cells (d).

The incidence set $\prec_{\mathcal{C}}$ specifies the topological structure of a mesh using the binary *incidence* relationship. The incidence relationship is a partial order between cells in $\mathcal{C}$, i.e., cells c and e are incident (i.e., $c \prec_{\mathcal{C}} e$) if one is located on the boundary of the other, i.e., $c \in \partial(e)$ ($dim(c) < dim(e)$) or $e \in \partial(c)$ ($dim(e) < dim(c)$) where $\partial()$ represents the boundary of the cell c. When k-cell c is on the boundary of m-cell e, then c is called a *k-face* of e and e is a *m-coface* of c or just *co-boundary* of c. When $k = m - 1$ then c is an *immediate boundary* face of e and e is an *immediate co-boundary* (coface) of c. Cells c and e ($k = m \neq 0$) are *p-adjacent* if they share a p-face. Two vertices v_1 and v_2 ($k = m = 0$) are adjacent if they share an edge. The adjacent relation can be expressed as nested application of the co-boundary and boundary relations, i.e., to find k-adjacent cells to the m-cell c, first we need to find all its boundary k-faces and then for each k-face find its m-cofaces.

The geometric embedding Γ maps each cell in $\mathcal{C}$ to its corresponding geometric realization such that $(c_1 \prec_{\mathcal{C}} c_2) \Leftrightarrow (\Gamma(c_1) \subset \Gamma(c_2) = \bigcup_{c \in \partial(c_2)} \Gamma(c))$. For instance, the geometric realization of vertices are their coordinates.

$\mathcal{F}$ contains a set of (partial) functions which assign data

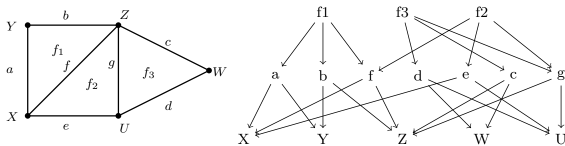

Figure 1: A 2D triangular unstructured mesh (left) and its incidence graph (right).

values to each cell. For instance, the function f_i assigns a value of type τ to each i-cells: $f_i : C^i \to \tau$ where C^i refers to i-cells.

We can represent mesh topology using *Incidence Graph (IG)* such that the nodes of the graph represents the cells of the mesh and the links between two nodes encode the incidence relationship. Figure 1 shows a simple 2D simplicial mesh (left) and it topological structure (right) as a incidence graph [13]. The incidence graph in Figure 1 shows that the boundary and co-boundary a cell can be extracted by following the links in the graph starting from the cell downward and upward, respectively. The immediate boundary and co-boundary cells of a cell are its children and parents. For instance, the immediate boundary and co-boundary of the edge d are $\{U, W\}$ and $\{f_3\}$, respectively and the boundary and immediate co-boundary of face f_3 are $\{d, c, g, W, U, Z\}$ and $\{\}$, respectively.

For detailed definitions, we refer the interested reader to [13], Chapter 3 of [3], and the references therein.

3. RELATED WORK

Stencil string specifies a subset of neighboring vertices of a given vertex on a structured mesh. The information of the neighbors are later used by numerical approximation algorithms to compute a weighted solution for the vertex. In other words, the stencils determine the region of influence of numerical algorithms. For example, commonly used neighborhood sizes by algorithms are three, five, and twenty five neighbors for 1D, 2D, and 3D structured meshes, respectively. The neighbors in a structured mesh can be easily specified by index arithmetic, e.g., the four neighbors of the point (i, j) in a 2D structured grid. In systems such as *Pochoir*, the neighboring vertices are explicitly listed [15]. Figure 2 (left) shows four neighboring vertices of the red vertex on a 3D structured mesh (5-point stencil).

In his thesis, Berti generalized the idea of stencils for unstructured meshes by abstracting topological structures of mesh algorithms as an *incidence sequence*. For instance, if the incidence sequence of a mesh algorithm is 010 (or alternatively shown as *vertex-edge-vertex*) w.r.t. an unstructured mesh, then it means any calculation for a vertex v needs to have access to all adjacent vertices of v (i.e., vertices sharing an edge with v) [3]. The snippet below shows how the stencil string 010 can be used to computes the data for each vertex v_1 as the sum of the data of its adjacent vertices (v_2).

```
forall vertices v1
  forall edges e incident to v1
      forall vertices v2 incident to e
         result[v1]+=data[v2]
```

Figure 2 (right) depicts 010 stencil for the red vertex in a 2D unstructured mesh. As it can be seen, the stencil string is implemented as nested for loops. Furthermore, Berti formally defined the concept of *incidence hull* as the union of all cells expressed by a stencil string. Gridfields uses stencil string in the same manner [7].

The proposed neighborhood expression in this paper extends the stencil string in [3]. The new neighborhood expression offers several advantages to the stencil string notation: it uses topological abstraction rather than dimensions, it can return two types of neighborhoods (i.e., hull or halo), and it is able to filter the intermediate results.

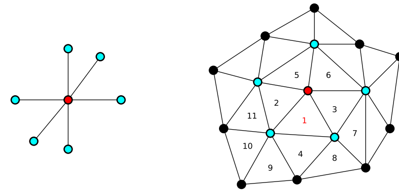

Figure 2: 5-point stencil for a vertex in 3D structured mesh (left) and adjacent vertices of the red vertex represented by 010 stencil in a 2D unstructured mesh (right).

The other related area to this research is *graph databases*. We use graph database *Neo4j* to implement the proposed neighborhood expression. Neo4j is a popular open source graph database implemented in Java which is operational since 2003. Neo4j is schema-free, supports full ACID transactions, and provides implementations for several graph algorithms out of the box. Neo4j uses *property graph* data model, i.e., data is stored in nodes and relationships of a multi-graph with pairs of key-value properties. Neo4j graphs can be queried using either its declarative query language (known as *Cypher*) or its Core Java API. We refer the reader to Neo4j documentation for further details on Neo4j and Cypher [1, 14].

The reason to choose graph databases is that they allow implementing the general purpose mesh data model (see Section 5.1). The reasons for choosing Neo4j rather than other graph databases such as *DEX* (implemented in C++) are two folds, namely, high performance on graph traversal queries and supporting spatial data. Recent research show that the performance of Neo4j drastically improved since its early version. The argument is supported by performance evaluation of several graph databases (e.g., such as Neo4j, DEX, orientDB, etc.) using data ingestion, traversal, non-traversal, and manipulation queries. Neo4j outperforms the other systems in traversal queries (the main focus of this paper) [8, 9, 11, 10]. Moreover, Neo4j is shown to have close performance to graph-processing frameworks such as *GraphLab* and *Giraph* [5]. The spatial layer is crucial for several mesh operations. This is beyond the scope of this paper and will not be covered here.

4. TOPOLOGICAL NEIGHBORHOOD EXPRESSION

The basic idea of the neighborhood expression is to use topological abstraction for expressing neighborhood traversal, i.e., to express a neighborhood as nested application of boundary, co-boundary, and adjacency functions. We define

these functions as follows:

Boundary Function. it is represented as $b(c, k)$ for a given cell c and it returns the boundary elements of dimension k ($k < dim(c)$) of the cell c:

$$b(c, k) = \{e|\ (e \in \partial(c))\ \wedge\ (dim(e) = k)\}$$

For a cell set C, it is defined as the union of k-cells on the boundary of each member of the set.

Immediate Boundary Function. it is represented as $ib(c)$ for a cell c and it returns the immediate boundary of c:

$$ib(c) = \{e|(e \in \partial(c)) \wedge (dim(e) = dim(c) - 1)\}$$

Co-Boundary Function. it is represented as $cob(c, k)$ for a given cell c and it returns the co-boundary of dimension k ($k > dim(c)$) of cell c:

$$cob(c, k) = \{e|\ c \in \partial(e)\ \wedge (dim(e) = k)\}$$

For a cell set C, it is defined as the union of k-cells on the co-boundary of each member of the set.

Immediate Co-Boundary Function. it is represented as $icob(c)$ for a given cell c and it returns the immediate co-boundary of c:

$$icob(c) = \{e|\ c \in \partial(e) \wedge (dim(e) = dim(c) + 1)\}$$

Adjacency Function. it is represented as $adj(c, k)$ for a given cell c and it returns the k-adjacent cells to the cell c, i.e., cells which share a k-face with c. Formally,

$$adj(c, k) = \{e|\ (\exists s : s \in (\partial(c) \cap \partial(e))) \wedge dim(s) = k\}$$

Note that $dim(c) = dim(e)$.
The definition of the adjacency relation for vertices is defined as follows:

$$adj(v) = \{c|\ \exists e : (dim(e) = 1) \wedge (v \prec e) \wedge (c \prec e)\}$$

where v and c are vertices. Note that in this case the function has only one argument.

4.1 Basic Neighborhood Expression

The basic neighborhood expression is semantically equivalent to the stencil string. The difference is that instead of using dimensions it uses the topological functions defined in the previous section. This means that arbitrary cell neighborhoods can be expressed as nested applications of the functions. Assuming the set C containing all initial elements which we would like to traverse their neighborhood (a.k.a. *seed*), the basic expression is defined as:

$$nex = e_n(e_{n-1}(\cdots e_1(\ e_0(C, k_0), k_1) \cdots, k_{n-1}), k_n)$$

where nex is the *neighborhood expression*, e_i is a (immediate) boundary, (immediate) co-boundary, or adjacency relationships, $0 \leq k \leq d$ (d is the mesh dimension), k_i shows an optional dimension argument for the functions, and $0 \leq i \leq n$. The expression repeatedly applies the functions to explore the neighborhood of the cells in the set C. Functions such as b needs a dimension parameter which specifies the dimension of the target boundary element. For the adjacency function, adj means p-cells which share a $(p-1)$-face with a given p-cell in the seed set.

The nex is a functional expression, i.e., its functions are applied from innermost to outermost, i.e., e_0 is applied to the seed set C and produces *layer* zero result L_0, e_1 is applied on L_1 and stored as *layer* one result L_1 and so on.

For instance, the stencil 010 (adjacent vertices of the seed vertices) can be expressed as $icob(ib(V))$ or simply $adj(V)$, where V is the seed set. As another example, a common stencil on 2D triangular mesh is 202, i.e., neighboring 2-cells which share at least one vertex with the seed 2-cell. In Figure 2 (right), the stencil for the triangle number 1 as seed contains triangles number 2 to 11. The stencil can be expressed as $cob(b(F, 0), 2)$ (or $icob(icob(ib(ib(F)))))$ where F is the seed set. The stencil 0102 can be succinctly expressed as $cob(adj(V), 2)$ with V as the seed set.

The final result of the expression can be either the elements in the last layer or union of elements in all intermediate layers. In the latter case, the result forms a *stencil mesh* [7]. We define these two types of outputs as follows:

Halo Neighborhood. A halo neighborhood w.r.t. a given neighborhood expression nex contains the cells of the last layer only, i.e., before computing elements of the current layer it removes cells from the previous layer:

$$halo(C, nex) = L^n$$

Hull Neighborhood. A hull neighborhood w.r.t. a given neighborhood expression nex contains the union of elements in all intermediate layers of the nex evaluation:

$$hull(C, nex) = \bigcup_{i=0}^{i=n} L^i$$

4.2 Advanced Neighborhood Expression

The neighborhood expression from Section 4.1 has enough abstraction w.r.t. mesh topology. However, in comparison to stencil string, it does not offer anything new. Often applications need to filter the intermediate results of the traversal while searching the neighborhood. This is not possible with stencil string. Thus, we further extend the basic neighborhood expression to allow filtering of cells in intermediate layers using predicates on mesh components (e.g., field values, geometry, etc.). To this end, we extend the definition of the functions as follows:

Extended Boundary Function. It is represented as $b(C, k, p)$. It computes the k-cells in the boundary of each element in C and then it checks if the boundary elements satisfies the predicate p. It returns elements in the boundary which satisfy p. It is formally defined as follows:

$$b(C, k, p) = \bigcup_{c \in C} \{e|(e \in \partial(c)) \wedge (dim(e) = k) \wedge p(e)\}$$

where $p(e)$ means that the cell e satisfies p.

Extended Immediate Boundary Function. It is represented as $ib(C, p)$. For each cell c in C, it computes its immediate boundary and applies the predicate p on each cell in the immediate boundary of c. It returns cells in the immediate boundary which satisfy the predicate. It is formally defined as follows:

$$ib(C, p) = \bigcup_{c \in C} \{e|(e \in \partial(c)) \wedge (dim(e) = dim(c) - 1) \wedge p(e)\}$$

Extended Co-Boundary Function. It is shown as $cob(C, k, p)$. For each cell c in C, it computes its co-boundary elements of dimension k and checks if they satisfies the predicate p. It returns co-boundary cells which satisfy the predicate. It is formally defined as follows:

$$cob(C, k, p) = \bigcup_{c \in C} \{e|(c \in \partial(e)) \wedge (dim(e) = k) \wedge p(e)\}$$

Extended Immediate Co-Boundary Function. It is represented as $icob(C, p)$. For each cell c in C, it computes its immediate co-boundary and applies the predicate p on each cell in the immediate co-boundary of c. It returns cells in the immediate co-boundary which satisfy the predicate. $icob(C, p)$ is formally defined as follows:

$$\bigcup_{c \in C} \{e | (c \in \partial(e)) \wedge (dim(e) = dim(c) + 1) \wedge p(e)\}$$

Extended Adjacency Function. It is represented as $adj(C, p)$. For each p-cell c in C, it computes its $(p-1)$-adjacent cells and checks the predicate p. It returns $(p-1)$-adjacent cells which satisfy the predicate. $adj(C, p)$ is formally defined as follows:

$$\bigcup_{c \in C} \{e | (\exists s : s \in (\partial(c) \cap \partial(e))) \wedge (dim(s) = k) \wedge p(e)\}$$

With the definitions above, the new functional neighborhood expression is as follows:

$$nex = e_n(e_{n-1}(\cdots e_1(\ e_0(C\ , k_0, p_0), k_1, p_1) \\ \cdots, k_{n-1}, p_{n-1}), k_n, p_n)$$

where k_i and p_i are the (optional) dimension and the predicate arguments, respectively. The predicate p_i is defined on properties of cells returned by e_i, e.g., predicates on data fields or geometric features such as length, area, volume, etc.

For instance, the expression $icob(ib(T, f < 24.0))$ (or equivalently $adj(T, 2, f < 24.0)$) finds 2-adjacent 3-cells using 3-cells in T as seed set. The predicate selects only 2-cells where the value of the field f is less than 24.0.

Such a functional notation with many nested parentheses can become very tedious to read/write. Thus, we propose a notation inspired by *XPath* [2]. We use slash to separate the topological functions and brackets to express predicates. Assuming C as the seed set, the previous nex can be written as follows:

$$nex = e_0(k_0)[p_0]/e_1(k_1)[p_1]/\cdots/e_{n-2}(k_{n-2})[p_{n-2}]/ \\ e_{n-1}(k_{n-1})[p_{n-1}]/e_n(k_n)[p_n]$$

Note that k_i and p_i are optional and the seed set is not present in the expression. In the next Section, we show how to specify the seed.
For instance, the expression from the previous example can be written as $ib[f < 24.0]/icob$ using T as seed.

In comparison to the functional form, the evaluation of XPath-like neighborhood expression is done left-to-right, i.e., first applying e_0 to the seed set C and filtering the result using p_0, then applying e_1 to the result from layer zero and so on. More formally, the elements in each intermediate layer is computed as follows:

$$nex = \begin{cases} L^0 = \bigcup_{c \in C} \{e | e \in e_0(c) \wedge p_0(e)\} & i = 0 \\ L^1 = \bigcup_{c \in L^0} \{e | e \in e_1(c) \wedge p_1(e)\} & i = 1 \\ \cdots & \cdots \\ L^n = \bigcup_{c \in L^{n-1}} \{e | e \in e_n(c) \wedge p_n(e)\} & i = n \end{cases}$$

where L^i represents elements in the layer i and $p_i(e)$ means that the predicate p_i holds for cell e.

5. IMPLEMENTATION

We implemented a set of algebraic mesh operators which we call *AMQL* (Algebraic Mesh Query Language). The operators are declarative meaning that we only need to describe the information need and the AMQL engine will figure out how to find it. Two of AMQL's operators use the neighborhood expression, namely, the *neighbors* and *self-regrid* operators. Other operators are described in [13]. In the rest of this section, first we discuss the data model which we use to store unstructured meshes as graphs and then we explain the neighbors and self-regrid operators.

5.1 Graph Data Model for Meshes

In Section 2, we discussed that incidence graph can store connectivity information of unstructured meshes. The graph model allows us to find boundary, co-boundary, and adjacency relationships of cell using graph traversal.

The IG does not encode information about the data fields and geometric embedding of cells. Many mesh application domains have operations which needs to manipulate fields and geometry data. Furthermore, the IG stores only incidence relationship and the adjacency relationship needs to be computed on demand. Neighborhood queries use adjacency information extensively. Adjacency based data structure are proved to be more efficient [4].

Based on the above observations, we extend the IG model to a general purpose mesh model such that nodes in the graph contains data fields and geometries and each node stores information about its adjacent nodes. Figure 3 shows the graph data model for a linear 3D unstructured mesh. The nodes in the graphs represent the cells, e.g, vertex, edge, face, and body. $VField_i$, $EField_i$, $FField_i$, and $BField_i$ represent properties of vertices, edges, faces, and bodies, respectively. *Geom* is the geometric object of each node, i.e., *point* for vertex, *line-segment*, *polygon* for face, and *volume* for body. This allows us to compute geometric predicates such as length, area, volume, centroid, distance, etc. Furthermore, there are two types of topological relationships between nodes, namely, boundary (between cells of different dimensions) and adjacency (represented as self-loop in the Figure 3). These relationships allow us to navigate the topology using graph traversal algorithms.

5.2 Self-Regrid Operator

In domains such as climatology and oceanography, the regrid operator is used to transform data from source meshes to a target mesh [6]. The operator works in two steps: first, it assigns a set of target cells to each source cell (mapping step), then, it combines the data of the mapped cells to estimate data of the target cell (interpolation step) [7].

The *self-regrid* operator is a special case of the regrid operator (i.e., the source and the target meshes are the same) where the goal is to estimate the data of a cell using its neighbors information. The mapping step uses topological or geometric neighborhood functions. The geometric mapping function, which assigns neighbors to a cell based on their geometric distance to the cell (e.g., k-nearest neighbors), is beyond the scope of this paper. The neighborhood expression introduced in this paper can be used as a topological mapping function in the self-regrid operator to assign neighboring cells to each given cell, e.g., assigns all adjacent vertices to each vertex.

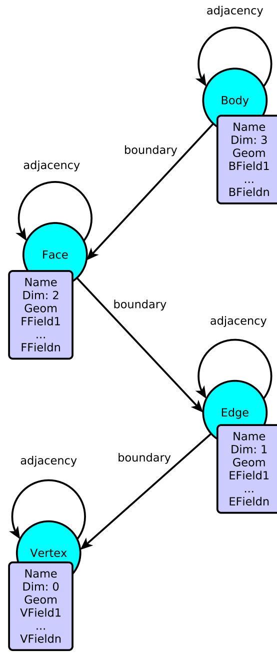

Figure 3: The graph data model for 3D linear meshes.

The notation for self-regrid operator is:

$$regrid(M,\ i,\ nex,\ agg_f = aggFunc(f))$$

The operator assigns cells defined by nex to each i-cell of M using the i-cell as the seed. Then, it applies the aggregation function $aggFunc$ on the field f of the mapped cells and store the result as a new field agg_f for the i-cell. The algorithm 1 shows how the self-regrid algorithm works. The algorithm loops over i-cells of the mesh M (line 7) and by using each i-cell as seed evaluate the topological functions from left-to-right. Th topological functions is stored in L and $L[i]$ refers to the ith function in the expression nex. For instance, if the topological function is ib and has a corresponding predicate, it first computes the result of the topological function and then for each cell in the result check the predicate. If there is no corresponding predicate, it just returns the result of the function (line 12-17). The value of field f of the mapped cells is combined to estimate value for the target cell (line 23-26).

For instance, the following self-regrid operator can be used to smooth temperature field on each vertex of the mesh M using information from its adjacent vertices. To find adjacent vertices we need to compute the neighborhood expression adj (or $icob/ib$). This is equivalent to the stencil string 010.

```
regrid(M, 0, adj , agg_temp=avg(temperature))
```

In more structured and readable form, we can write it as follows:

```
FOR VERTEX v IN M
MAP v TO neighbors(M, v , halo , icob/ib ) AS m
RETURN v, avg(m.temperature)
```

The FOR loop iterates over all vertices of the mesh M. For each vertex, the operator maps the vertex to the output set

Algorithm 1: Evaluation of the Regrid Operator $regrid(M,\ i,\ nex,\ agg_f = aggFunc(f))$

```
input : Mesh M, dimension i, field f, aggregation function
        aggFunc, output field name agg_f, and neighborhood
        expression nex
output: Mesh M with new field agg_f
1  L ← List of (co-)boundary functions extracted from nex;
2  P ← List of predicates extracted from nex;
3  MappedCells ← ∅;
4  Seed ← ∅;
5  S ← ∅;
6  fvals ← ∅;
7  while (there are i-cells in C) do
8      Seed ← next unused i-cell c;
9      k ← i;
10     for (j from 0 to length(L))) do
11         for (e in Seed) do
12             if (P[i] ≠ null) then
13                 I ← L[j](e);
14                 add cell c from I to S where P[i](c) holds;
15             else
16                 add ib(e) to S;
17             end
18         end
19         Seed ← S;
20         S ← ∅;
21     end
22     MappedCells ← Seed;
23     for (k-cell e in MappedCells) do
24         add e.f to fvals;
25     end
26     c."agg_f" = aggFunc(fvals) ;
27     MappedCells ← ∅;
28     fvals ← ∅;
29 end
```

m from the neighbors operator (using the MAP ... TO ... AS clause) and return the average of the temperature of the mapped cells. The dot notation is used to refer to the field temperature of a vertex, i.e., e.temperature.

5.3 Topological Neighbors Operator

The neighbors operator can create sub-meshes (a.k.a. stencil meshes). It is represented as $\mathcal{N}(M, E, ROI, nex)$ where its arguments are a mesh, seed set, the *Region Of Influence* (ROI), and a neighborhood expression, respectively. The *Region Of Influence (ROI)* is either *halo* or *hall*. The operator returns a set of cells by evaluating the expression nex on the seed set E w.r.t. to the ROI argument.
For instance, the hull mesh around a vertex v containing the vertex itself and all edges and faces can be expressed as $\mathcal{N}(M, \{v\}, hull, icob/icob/ib/ib)$.

Algorithm 2 shows how to construct result of the neighbors operator $\mathcal{N}(M, E, halo, nex)$. A similar algorithm can be written for the hull with small modification of the algorithm 2.

6. EXPERIMENTAL RESULTS

6.1 Experimental Setup

Experimental Design. We conducted experiments to evaluate the performance of the declarative self-regrid operator. The operator is implemented within AMQL which contains a collection of declarative operators for unstructured meshes implemented in Java. As explained in Section 3, the main reason to use Java is that Neo4j provides spatial data management (crucial for some of the operators) and either performs better or has close to existing graph

Algorithm 2: Algorithm Evaluation of Neighbors Operator $\mathcal{N}(M, E, halo, nex)$

```
input  : Mesh M, seed cells E, ROI = halo, and neighborhood
         expression nex
output: Sequence N containing all cells which are in the defined
         neighbourhood by nex
1  N ← ∅;
2  L ← List of (co-)boundary functions extracted from nex;
3  P ← List of predicates extracted from nex;
4  Seed ← E;
5  S ← ∅;
6  I ← ∅;
7  k ← dimension of element in E;
8  for (i from 0 to length(L)) do
9      for (e in Seed) do
10         if (P[i] ≠ null) then
11             I ← L[i](e);
12             add cell c from I to S where P[i](c) holds;
13         else
14             add ib(e) to S;
15         end
16     end
17     Seed ← S;
18     S ← ∅;
19 end
20 N ← Seed;
```

databases and frameworks such as DEX or GraphLab. We compare the performance of our implementation with *GrAL* (a C++ mesh library) [3] by measuring the execution time (this does not include time of building internal data structures or indexes for each system). The reason to use GrAL is that it uses a generic approach to meshes which enables it to express virtually any combinatoric query using domain specific language of iterators [3].

We use four smoothing field queries in the experiments. The queries differ in the length of the neighborhood expressions and use of predicates. This allows us to observe the effect of expression length and predicates on the performance.

Implementation Details. We run the experiments on a system with four cores (2.4 GHz processor) with 8GB of RAM and XUbuntu 12.10 operating system.

We use *ANTLR* to parse AMQL operators including the self-regrid [12]. The operators are implemented in Java and use Neo4j database facilities, e.g., storage, indexes, traversal framework, etc. However, the AMQL implementation is storage neutral, i.e., the implementation is abstracted and can be used with any other system which provides implementations for the abstract methods.

We implemented the self-regrid operator using both Cypher and Neo4j core Java API. We refer to these two implementations as AMQL_Cypher and AMQL_Java, respectively. In particular, the implementation of AMQL_Cypher translates each regrid query to a Cypher query. We report performance of each implementation .
We use GrAL as of 1.11.2014 and Neo4j 2.1.6. GrAL is compiled using gcc 4.6.3 with setting -03 which controls depth of template instantiation. GrAL implements each query separately in C++. We run each query ten times and average the response times over ten trials for each pair of (query,dataset).

Dataset. We use a real dataset from oceanographic domain [7]. The dataset contains a 2D triangular mesh where each vertex has two data fields, namely, temperature and bathymetry. The number of vertices, edges, and faces in the dataset are 20736, 39133, and 59884, respectively. To see how the systems perform with the data set size, we applied the subsetting operator from AMQL to divide the dataset to three smaller datasets with different size of vertices, i.e., 4862 (D1), 10270 (D2), and 20736 (D3). As it can be seen, the second dataset has (almost) twice the number of vertices as in the dataset two and the dataset three has twice the number of vertices as in the dataset two. The reason behind the subsetting is that all the queries needs to iterate over all the vertices in the datasets. This means the workload for each dataset is twice the previous dataset.

Queries. We use four field smoothing queries for the experiment. The smoothing operation is commonly used to smooth a noisy data field or a data field with missing values. The queries are as follows:

Q1. *Compute temperature of each vertex as average of the temperature of its adjacent vertices.*

In the Section 5.2, we showed the regrid operator for this query and its Cypher translation is as follows:

```
MATCH (p0:M)
WHERE p0.dim=0
WITH p0
MATCH (p0:M)<-[:ADJACENCY]-(p1:M)
RETURN p0.cid, avg(p1.temperature)
```

The *MATCH* clause is used for graph pattern matching. The first MATCH clause defines an iterator variable $p0$ on all nodes of mesh M. The WHERE clause filters $p0$ to vertices where *dim* property is zero. The *WITH* clause chains several smaller queries. In the query, the WITH only passes the vertices to the next part of the query. The second MATCH clause does a path matching in the graph by selecting all $p1$ nodes which has ADJACENCY relationship with $p0$. Finally, the RETURN clause returns the identifier of each vertex in $p0$ and average value of temperature over all correspondent $p1$. We refer the interested reader to Neo4j documentation for elaborate details on Cypher [1].

The GrAL C++ code implementing the same query consists of 10 lines of codes which uses underlying GrAL abstraction such as Cell-On-Cell iterators and mesh functions.

Q2. *Compute temperature of each vertex as average of the temperature of its adjacent vertices with the bathymetry field greater than 5.0.*

The query can not be expressed by a stencil. The query is equivalent to the neighborhood expression *adj[bathymetry > 5.0]* or *icob/ib[bathymetry > 5.0]*. The implementation of the query in the AMQL is as follows:

```
regrid(M, 0, adj[bathymetry > 5.0],
        agg_temp=avg(temperature))
```

The regrid operator above is translated to the the following Cypher query:

```
MATCH (p0:M)
WHERE p0.dim=0
WITH p0
MATCH (p0:M)<-[:ADJACENCY]-(p1:M)
WHERE p1.bathymetry>5.0
RETURN p0.cid, avg(p1.temperature)
```

The GrAL C++ code implementing the same query consists of 10 lines of codes.

Q3. *Compute temperature of each vertex as average of the temperature of vertices which are exactly two edges (2-hops) away from the vertex.*

This is equivalent to the stencil string 01010 and the neighborhood expression adj/adj (or $icob/ib/icob/ib$). The self-regrid operator pertaining to the query is:

```
regrid(M, 0, adj/adj,
        agg_temp=avg(temperature))
```

The regrid operator is translated to the following Cypher query.

```
MATCH (p0:M)
WHERE p0.dim=0
WITH p0 MATCH (p0:M)<-[:ADJACENCY]-(p1:M)
        <-[:ADJACENCY]-(p2:M)
RETURN p0.cid, avg(p2.temperature)
```

The GrAL C++ code implementing the same query consists of 15 lines of codes. The code uses several abstraction concepts from GrAL [3] and contains three nested FOR loops.

Q4. *Compute temperature of each vertex as average of the temperature of vertices which are exactly two edges away from it. Consider immediate adjacent vertices only if their bathymetry field is greater than 5.0.*

There is no stencil equivalent to this query. The query is equivalent to the neighborhood expression *adj[bathymetry > 5.0]/adj* (or *icob/ib[bathymetry > 5.0]/icob/ib*). The implementation of the query in AMQL is as follows:

```
regrid(M, 0, adj[bathymety > 5.0]/adj,
        agg_temp=avg(temperature))
```

The above regrid operator is translated to the following Cypher query.

```
MATCH (p0:M)
WHERE p0.dim=0 WITH p0
MATCH (p0:M)<-[:ADJACENCY]-(p1:M)
        <-[:ADJACENCY]-(p2)
WHERE p2.bathymetry>5.0
WITH p0, p2
MATCH (p2:M)
RETURN p0.cid, avg(p2.temperature)
```

Note that the complexity of the Cypher query grows with the length of the expression and the number of predicates. Moreover, some predicates such as geometric predicates can not be translated to Cypher.

The corresponding GrAL code for the query consists of 18 lines with three nested FOR loops.

6.2 Performance Evaluation

Figure 4, 5, 6, and 7 show the results of the **Q1**, **Q2**, **Q3**, and **Q4** queries. Clearly, the GrAL implementation outperforms AMQL in all the queries except **Q3**. A closer look on the performance data of the **Q1**, **Q2**, and **Q4** queries shows that GrAL on average is 150 (570), 140 (565), 500 (60) percent faster than AMQL_Java and AMQL_Cypher, respectively. However, in **Q3**, GrAL is on average 15 percent slower than AMQL_Java and 140 percent faster than AMQL_Cypher.

The performance of AMQL_Java increases by increasing the length of the neighborhood expression (see Figures 4 and 6). Its performance on Q3 even outperforms GrAL. This means that the traversal framework of Neo4j is very efficient on long expression. However, increasing the length and adding predicates cause a drastic increase in the performance of AMQL_Java (see Figures 6 and 7). This means that the traversal framework of Neo4j Java API does not perform well on a complex neighborhood expression with long length and predicates.

In comparison to AMQL_Java, the performance of AMQL_Cypher is better on longer expressions with predicates (see Figures 7). The reason is that we need to apply the traversal framework several times while evaluating an expression with predicates. Furthermore, the evaluation of predicates is done using Java code. However, AMQL_Cypher breaks down the query to shorter path and applies the predicates directly on Neo4j (which is faster than running on Java).

We conclude that both Cypher and Neo4j Java API should be used in implementing of the the expression depending on length of the expression and usage of predicates.

We observe that by increasing the length of the neighborhood expression and adding predicates the performances of AMQL and GrAL get closer (see Figure 6 and 7).

A common pattern in the performance of the both systems is that the response times increase linearly with the number of vertices. More precisely, for each pair of (system, query) the execution time on D2 is (almost) twice the execution on D1 and the execution time on D3 is (almost) twice the execution on D2.

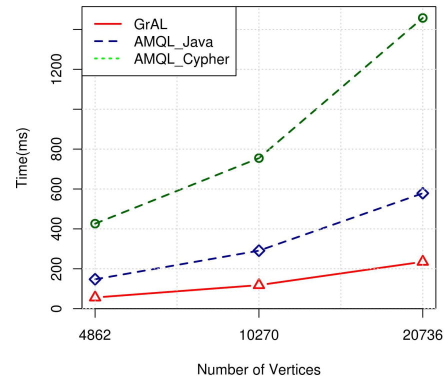

Figure 4: Performance of Q1 on AMQL and GrAL.

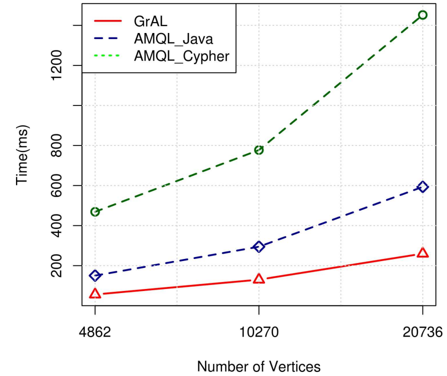

Figure 5: Performance of Q2 on AMQL and GrAL.

We believe that the poor performance of AMQL in comparison to GrAL has the following reasons. First, AMQL provides a generic solution which can accept any neighborhood expression as input while the GrAL implementations are query specific, i.e., any changes in the query requires changes in the implementation. The generic solution offers a declarative way of expressing the self-regrid but it has a cost which is the query parser overhead. Moreover, we use Neo4j's transactions in the implementation which introduce significant overhead. Also, in comparison to GrAL which uses a light and pure topological data structure, AMQL uses

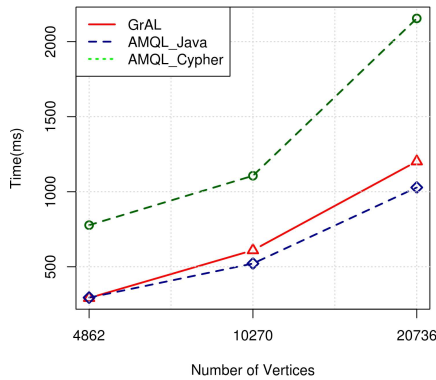

Figure 6: Performance of Q3 on AMQL and GrAL.

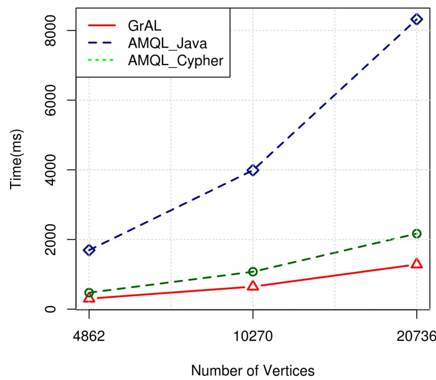

Figure 7: Performance of Q4 on AMQL and GrAL.

a general purpose mesh data model which contains the complete mesh information. This introduces further query processing overhead. Finally, it is known that Java language has inherent performance inefficiencies compare to C++.

To sum up, AMQL implementation works better on long neighborhood expression without predicates. In terms of expressiveness, the expression described in this paper is more expressive than the stencil string. Furthermore, it offers declarative querying (i.e., shorter and more readable than C++) and allows persisting of the computed data (i.e., the result of the regrid can be stored as a new field in the input mesh).

7. CONCLUSIONS AND FUTURE WORK

We presented a topological neighborhood expression which is more expressive than the stencil string. The implementation of the expression is declarative and generic. However, compare to a query-specific implementation in C++ it performs poorly (except on very long expressions).

In the future, we would like to measure the cost of the operator w.r.t. to the total cost of the queries and improve the operator implementation. Also, we would like to implement the operators on top of a graph database (framework) written in C++ such as DEX or GraphLab and repeat the experiments. We also want to use the expression in a structured query language for unstructured meshes similar to the example in the Section 5.2.

8. ACKNOWLEDGMENTS

The authors would like to thank Dr. Guntram Berti for his help in implementing the queries in GrAL library.

9. REFERENCES

[1] Neo4j - the world's leading graph database, Viewed December 2014.

[2] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernandez, M. Kay, J. Robie, and J. Siméon. Xml path language (xpath). *World Wide Web Consortium (W3C)*, 2003.

[3] G. Berti. *Generic software components for Scientific Computing*. PhD thesis, BTU Cottbus, 2000.

[4] D. Canino, L. De Floriani, and K. Weiss. Ia*: An adjacency-based representation for non-manifold simplicial shapes in arbitrary dimensions. *Computers & Graphics*, 35(3):747–753, 2011.

[5] Y. Guo, M. Biczak, A. L. Varbanescu, A. Iosup, C. Martella, and T. L. Willke. Towards benchmarking graph-processing platforms. *Poster at Supercomputing*, 2013.

[6] H. Hinterberger, K. A. Meier, and H. Gilgen. Spatial data reallocation based on multidimensional range queries. a contribution to data management for the earth sciences. In *Scientific and Statistical Database Management*, pages 228–239. IEEE, 1994.

[7] B. Howe. *Gridfields: model-driven data transformation in the physical sciences*. PhD thesis, Portland, OR, USA, 2007. AAI3255425.

[8] S. Jouili and V. Vansteenberghe. An empirical comparison of graph databases. In *Social Computing (SocialCom), 2013 International Conference on*, pages 708–715. IEEE, 2013.

[9] V. Kolomičenko, M. Svoboda, and I. H. Mlýnková. Experimental comparison of graph databases. In *Proceedings of International Conference on Information Integration and Web-based Applications & Services*, page 115. ACM, 2013.

[10] P. Macko, D. Margo, and M. Seltzer. Performance introspection of graph databases. In *Proceedings of the 6th International Systems and Storage Conference*, page 18. ACM, 2013.

[11] R. C. McColl, D. Ediger, J. Poovey, D. Campbell, and D. A. Bader. A performance evaluation of open source graph databases. In *Proceedings of the first workshop on Parallel programming for analytics applications*, pages 11–18. ACM, 2014.

[12] T. Parr. *The definitive ANTLR reference: building domain-specific languages*. Pragmatic Bookshelf, 2007.

[13] A. Rezaei Mahdiraji. Toward unstructured mesh algebra and query language. In *Proceedings of the 2014 SIGMOD PhD Symposium*, pages 16–20. ACM, 2014.

[14] I. Robinson, J. Webber, and E. Eifrem. *Graph databases*. " O'Reilly Media, Inc.", 2013.

[15] Y. Tang, R. Chowdhury, C.-K. Luk, and C. E. Leiserson. Coding stencil computations using the pochoir stencil-specification language. In *3rd USENIX Workshop on Hot Topics in Parallelism (HotPar'11)*, 2011.