Imperial College London
Department of Computing

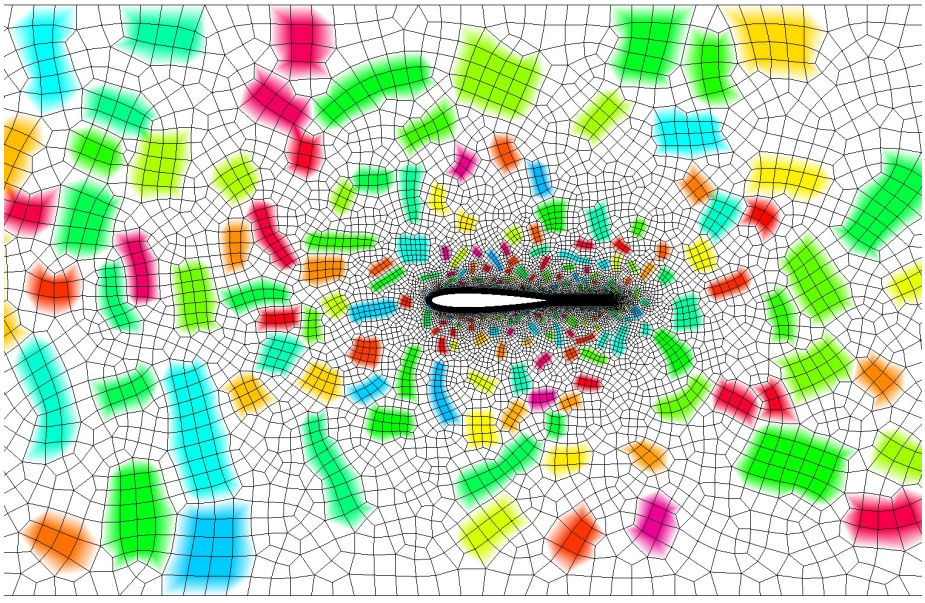
MEng Final Project Report

Crystal: Identifying and leveraging structure in quad meshes

Wael Al Jishi <wa910@imperial.ac.uk>

Supervisor:
Professor Paul Kelly

Second Marker:
Dr David Ham



Abstract

It is widely believed that the performance of computations over structured meshes is superior to those over meshes with no inherent structure. Unstructured meshes, however, are more ubiquitous in practice, necessitated by the demands of many real world applications.

Existing methods are typically applied at the mesh generation stage, altering the geometric and topological arrangement of the mesh in order to tease out fragments of structure. Altering the mesh in this way may not always be acceptable, or indeed possible, and it furthermore breaks the abstraction between the mesh model and its representation.

To bridge this gap, we present Crystal, a group of algorithms for *extracting* regions of uniform quadrilateral structure in an unstructured mesh and reorganizing the mesh *representation* to *expose* said structure in order to enable efficient *exploitation* of the underlying structure, all whilst preserving the mesh model.

To this end, we demonstrate how a loop executing a compute kernel may be transformed to leverage the detected structure. We evaluate our runtime performance on a non-linear airfoil lift calculation, and find with promising results. Our performance on a fully-structured mesh is reduced by 5%, however for typical cases with scope for improving data locality we gain an 11% improvement, and manage to achieve up to a two-fold increase in performance on meshes with bad numbering.

A word of thanks

To Professor Paul Kelly, Fabio Luporini, and Gheorghe-Teodor Bercea (Doru) for their outstanding efforts, tireless dedication and invaluable discussions.

To Dr. David Ham and various members of the Software Performance Optimisation group for their much appreciated help and benevolent contributions to discussions.

To my family for their unconditional love and unwavering support.

To my friends for their warm-hearted presence and delightful company.

Contents

Chapter 1

Introduction

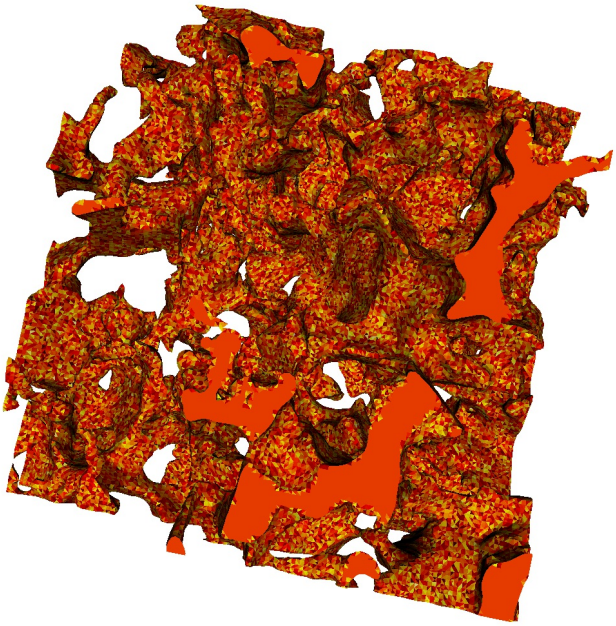
1.1 Motivation

Scientific computing is a large research area touching on various areas in the scientific community as well as in various industries. An integral part of it is concerned with algorithms and techniques which operate on a mesh representation of a model, typically modelling physical phenomena such as the motion of fluids. In essence, a mesh is a discretisation of a field that serves as an approximation of the true underlying field, making it amenable to numerical computation methods. The meshes are then the target of many numerical computations; evaluating partial differential equations is a classic example, which uses techniques such as the finite-element and finite-volume[1] methods, the latter used widely on unstructured meshes.

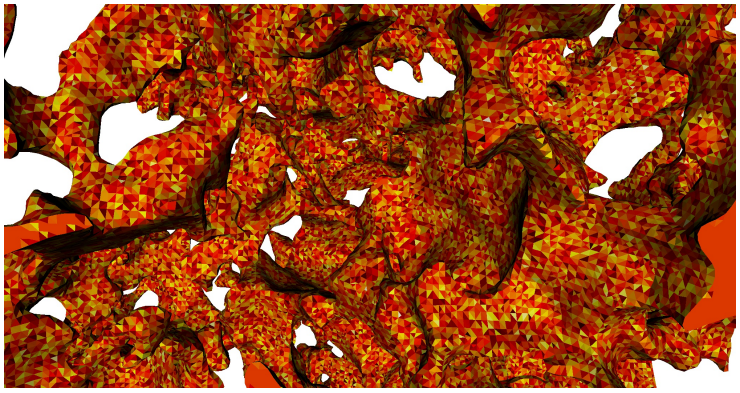
Meshes may be represented using various data structures such as the winged-edge, the half-edge, or the face-based representation. Representations often rely on encoding relationship between elements in some form of explicitly-defined mapping between mesh elements (chapter 2).

[1] See [29] by Versteeg and Malalasekera for an excellent introduction to computational fluid dynamics and the finite-volume method.

As an example, below is a mesh representing a Micro CT image of a Berea sandstone [2].



A close-up image reveals that it is composed of many[3] of small triangles forming tetrahedra:



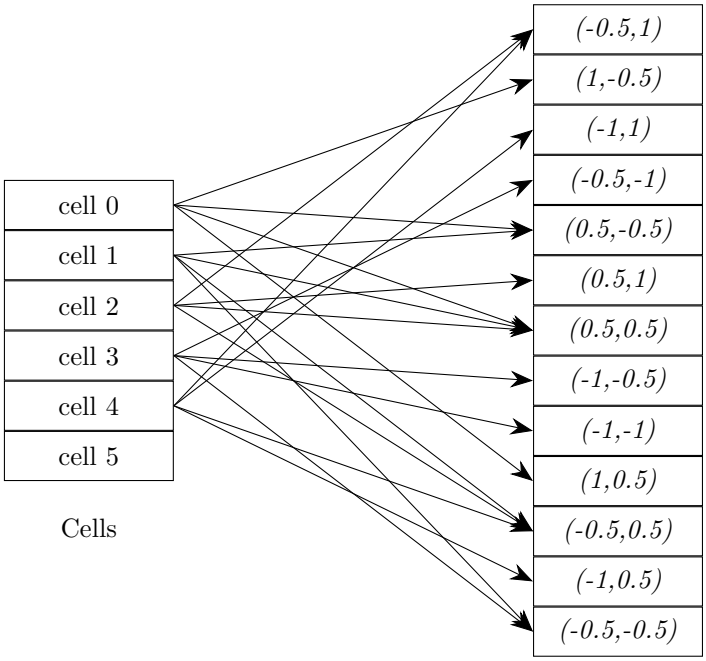
[2]The Berea-8 mesh was kindly provided by Dr. Gerard Gorman
[3]Millions, in fact

1.1.1 Representing non-structure

Suppose that we wish to compute the surface area of this mesh. For this two pieces of information are required:

- The position of each vertex in space.
- The vertices that form each face; equivalently, the vertex connectivity.

This information may be represented generally with a data structure similar to the following:



Notice how adjacent cells access vertices in a random fashion[4]. This is detrimental to performance for various reasons:

1. They exhibit weak spatial locality, a property which most modern CPU caches bank on to attain higher performance in IO bound applications, which may manifest through decisions regarding cache replacement strategies or data pre-fetching.
2. The indirection maps compete with valuable data for memory access as well as memory storage, both resulting in degraded cache performance.

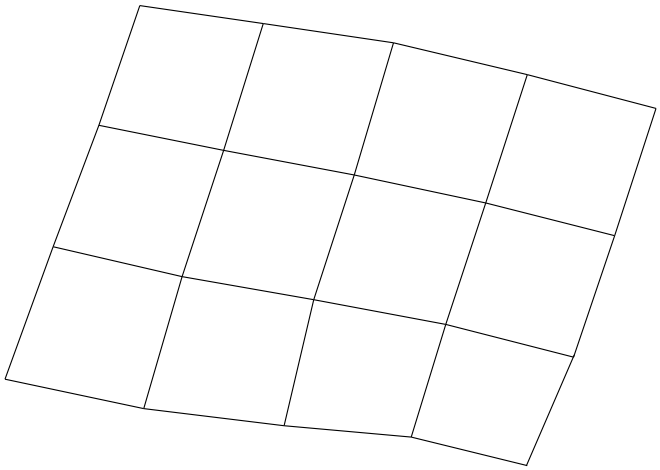
[4]Or at least random looking. The point is that there is clearly no structure.

3. Looking up addresses, as opposed to computing them directly, will typically prohibit or limit the scope of compiler-performed optimizations, not least vectorizations.

Numerous strategies have been devoted to dealing with this problem, notably applying a space filling curve to obtain a more favourable numbering, with closer elements tending to have closer numberings. While the space filling curve most certainly improves cache locality, it does not make use of more obvious structure that may exist. A mesh that is irregular and unstructured on the whole may contain subregions of high regularity and uniform structure, whose regularity may be locally exploitable in a more direct manner, for potentially higher performance gains!

1.1.2 Representing structure

To contrast, let us consider the following fully structured mesh:



As the vertices form a two-dimensional lattice, we can store their coordinates in a two-dimensional array such as the following:

(2.0,8.9)	(3.9,8.4)	(5.9,7.9)	(7.9,7.4)	(9.9,6.9)
(1.4,6.8)	(3.3,6.4)	(5.3,6.0)	(7.3,5.5)	(5.3,4.9)
(0.7,4.9)	(2.7,4.5)	(4.7,4.1)	(6.7,3.6)	(4.6,3.1)
(0.0,2.9)	(2.1,2.6)	(4.2,2.4)	(6.2,2.0)	(8.2,2.5)

The cells are now defined implicitly, one between each four vertices forming a quad. We need not explicitly store the mesh connectivity and computations can access the data directly without going through indirections.

1.2 Contributions

We present Crystal, a group of algorithms for *extracting* regions of regularity in a mesh, reorganizing the mesh to *expose* said structure in order to enable efficient *exploitation*. The Crystal algorithms are also embodied in a working prototype implementation.

We present a summary of our contributions, and discuss them in detail in what follows.

- Algorithms for extracting structure, and an evaluation of their competency.
- Methodologies for reorganising the data layout to expose the structure, and a discussion about their relative merits.
- A scheme for exploiting the structure efficiently, and a performance evaluation of its effectiveness.

1.2.1 Extracting structure

We formally define a notion of vertex structure in a mesh, in particular the properties that the vertex-vertex adjacency must exhibit within a structured region. This covers the topics of

On this basis, a vertex structure extraction algorithm is devised and implemented. The algorithm traverses a constructed graph whose vertices correspond to the mesh's original vertices, and whose edges represent the vertex-vertex adjacency relation. Each structured region is "grown" from a given seed vertex, ensuring with each step that the extracted region is a well-formed structured region (chapters 4 through 6). The illustration on the title page of this thesis showcase the result of structure detection for our tools.

We also define the notion of a structured region for cells and edges, and develop algorithms for deriving said structure using previously extracted vertex structure.

1.2.2 Exposing structure

We describe how the underlying data storage layout must be reorganised to reflect the extracted structure and facilitate its exploitation. This must take into account elements within the unstructured regions, ensuring that accesses via the modified maps remain consistent with the originals (chapter 7).

1.2.3 Exploiting structure

Using the information gathered from the extracted structure, we can transform the data storage layout so as to implicitly represent the adjacency relationships between various mesh elements. This alleviates the need to use indirection maps when running computations over a structured region, substituting them for a

small constant amount of meta-data per structured region that is used to define each region's dimensions and orientation. Computations over an unstructured region execute as normal using restructured versions of the original maps and data (chapter 8).

This allows us to transform a cell-to-nodes access operation from the following, which uses indirections to access the node data:

```
for (int cell_id = 0 ; cell_id < num_cells ; ++cell_id) {
  int node0 = cell_to_nodes[cell_id][0];
  int node1 = cell_to_nodes[cell_id][1];
  int node2 = cell_to_nodes[cell_id][2];
  int node3 = cell_to_nodes[cell_id][3];

  int node0_data = data[node0];
  int node1_data = data[node1];
  int node2_data = data[node2];
  int node3_data = data[node3];

  // Use data
}
```

To something like this, which computes the location of node data directly:

```
for (int row = 0 ; row < num_rows ; ++row) {
  for (int col = 0 ; col < num_cols ; ++col) {
    int node0 = row * nodes_per_row + col;
    int node1 = row * nodes_per_row + (col+1);
    int node2 = (row+1) * nodes_per_row + col;
    int node3 = (row+1) * nodes_per_row + (col+1);

    int node0_data = data[node0];
    int node1_data = data[node1];
    int node2_data = data[node2];
    int node3_data = data[node3];

    // Use data
  }
}
```

Chapter 2

Background

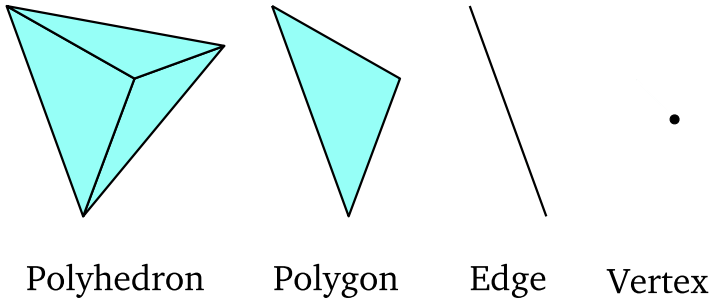
Before continuing further, we introduce briefly the main notions required to appreciate this work.

2.1 The mathematical mesh model

Meshes often model physical objects and phenomena. This is typically achieved through the discretization of a continuous model, such as the surface or volume of an object, in order to approximate its physical properties to a desired degree of precision.

The mesh model consists of a hierarchy of elements, which may include a subset the following:

- Polyhedra such as cubes or tetrahedrons
- Polygons *(also referred to as cells or faces)* such as triangles and quadrilaterals
- Edges
- Vertices *(also referred to as nodes)*



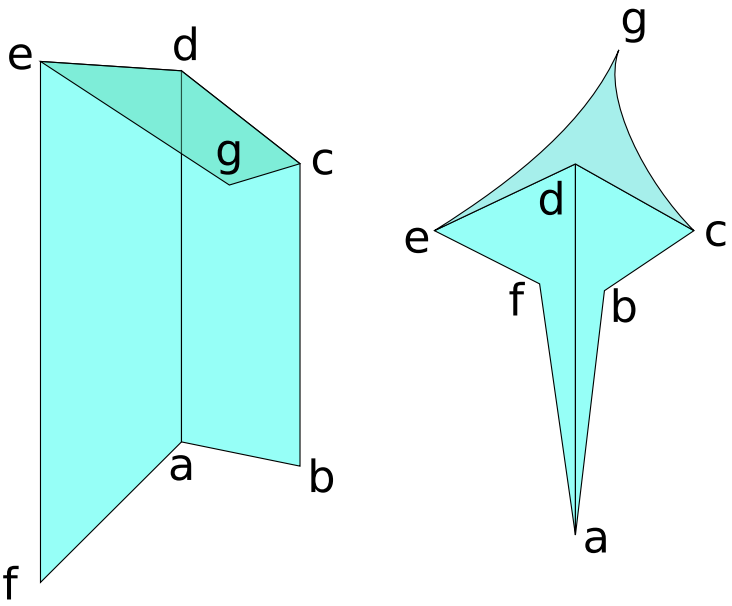


Figure 2.1: Despite having completely different geometric shapes and properties, the two meshes are topologically equivalent. The labels indicate corresponding vertices.

Each element in the above hierarchy is built-up from those below it. Thus, a polyhedron is assimilated by a set of polygons, a polygon is composed by a set of edges, and an edge joins two vertices.

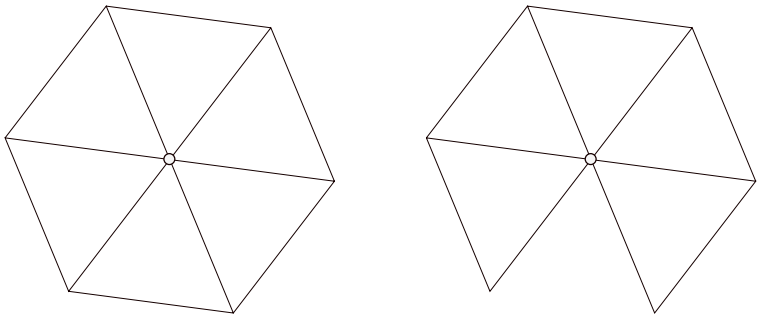
2.1.1 Geometry vs topology

There is a key distinction to make between the geometric and topological properties of a mesh.

Since meshes model a physical reality, the elements of a mesh may be spatially embedded: vertices are associated with points in space, and edges are formed as segments joining their two vertices. This affects *geometric* properties of the mesh, such as its surface area or volume.

On the other hand, the hierarchy of elements described above induces a mesh topology. This describes the connectedness of the mesh, that is to say how elements relate to one another. For instance, we may describe two vertices sharing an edge as *adjacent*, or two cells being sharing an edge as being *incident* on that edge.

In this work we concern ourselves solely with the topological structure of meshes, treating its geometry as arbitrary data that is associated with its respective elements (the position of a vertex for instance). Figure 2.1 illustrates the difference between the two concepts.



(a) A closed fan **(b)** An open fan

Figure 2.2

2.1.2 Manifold meshes

A mesh is a manifold if the following properties hold:

1. All edges are adjacent to either one or two faces.
2. All faces meeting at a given vertex must form either an open or a closed fan around that vertex (Figure 2.2).

Figure 2.3 demonstrates examples of non-manifold meshes. In this work we consider manifold meshes exclusively, and future mentions of 'mesh' shall implicitly refer to manifold meshes.

2.2 The mesh data structure

We describe how a mesh model is manifest at the data structure level. There are three general component types that can be identified:

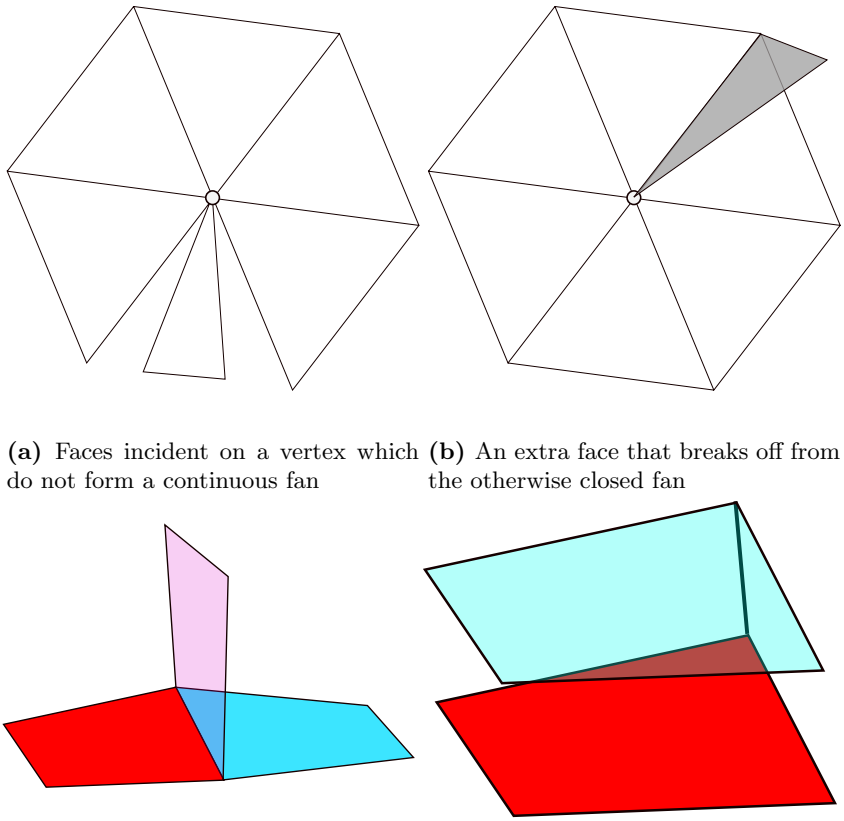
- Entity sets
- Associated data
- Relations between two entity sets

In the following sections, the examples shall refer to the mesh depicted in figure 2.4.

2.2.1 Entity sets

Each set represents a certain type of entity in the mesh, such as vertices or cells. Each element in a set is associated with a unique identifier. Integers are a common choice as an identifier for a couple of reasons:

- They need not be enumerated explicitly. All we need is the set cardinality and a starting index.



(a) Faces incident on a vertex which do not form a continuous fan

(b) An extra face that breaks off from the otherwise closed fan

(c) More than two faces incident on a single edge

(d) An edge with no incident faces

Figure 2.3: Examples of non-manifold meshes.

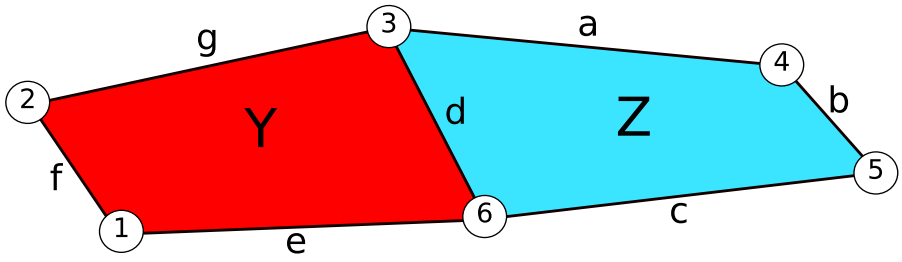


Figure 2.4: Example mesh with labelled elements.

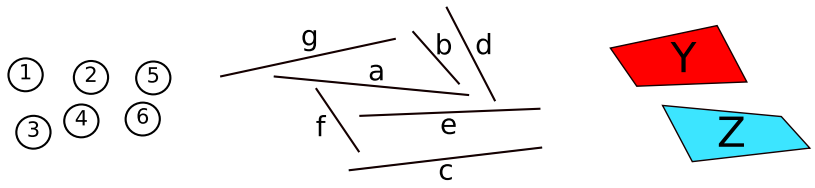


Figure 2.5: The entity sets of the mesh in figure 2.4. These are (from left to right) the vertices, edges, and cells.

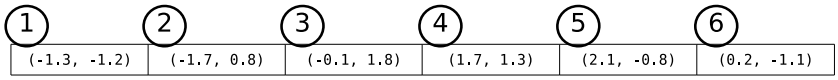


Figure 2.6: Coordinate data associated with the vertices of the mesh in figure 2.4.

- They are convenient for direct-indexed array accesses, as well as for more general indexing methods.

See figure 2.5 for examples.

2.2.2 Associated data

Arbitrary data which is associated with elements of a particular entity set. For instance, spatial coordinates associated with each vertex. A typical representation is a flat array indexed by element identifier. This is the data over which we perform our computations and ultimately for preserving. Everything else is incidental. See figure 2.6 for an example.

2.2.3 Relation maps between two entity sets

Entity sets may have relations defined between them, a mapping from an element in a source set to one or more corresponding elements in the destination set. For instance, we may have an adjacency relation from the vertex set to itself, or an inclusion relation from the cell set to the vertex set. In a general unstructured mesh these relations must be explicitly stored, typically as an array indexed by the source element's identifier. See figure 2.7 for an example.

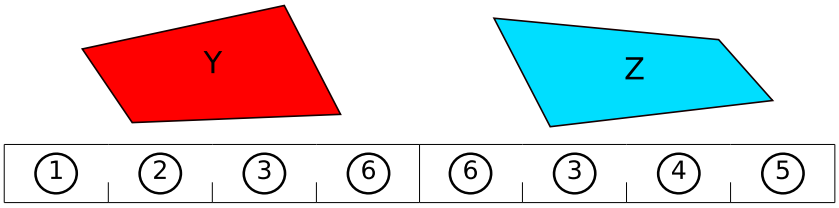


Figure 2.7: Inclusion relation from cells to vertices, as depicted in the mesh of figure 2.4.

2.3 The core-computation contract

Given a mesh model and its underlying representation, computation logic provided by an external user is to be executed. We refer to this as the *core-computation* so as to disambiguate it from other incidental processing, such as structure detection. Our contract to the user is described in what follows.

2.3.1 Given: operating set

We are given an entity set over which to operate, for example the set of edges or the set of cells. We refer to this entity set as the *operating set*. The core-computation consists of executing a computation for each element of the operating set. This is analogous to the *map* phase of the MapReduce programming model [7], though we restrict our usage of the term *map* to refer to relation maps.

2.3.2 Given: relation-map tree

We are given a tree structure defining which relation maps to use and how to access them. This is best explained through an example, illustrated in figure 2.8. The core-computation will, for each element in the operating set, gather all indexing variables as described by the relation-map tree.

2.3.3 Given: kernel function

We are given a kernel function specifying the computation logic, which is applied to each element in the operating set. It takes as arguments all gathered indexing variables, including that of the current element, and it has read and write access to the mesh's associated data. The access pattern of a kernel function is similar to that of a stencil computation, as defined by [26]:

> A stencil computation repeatedly updates each point of a d-dimensional grid as a function of itself and its near neighbours.

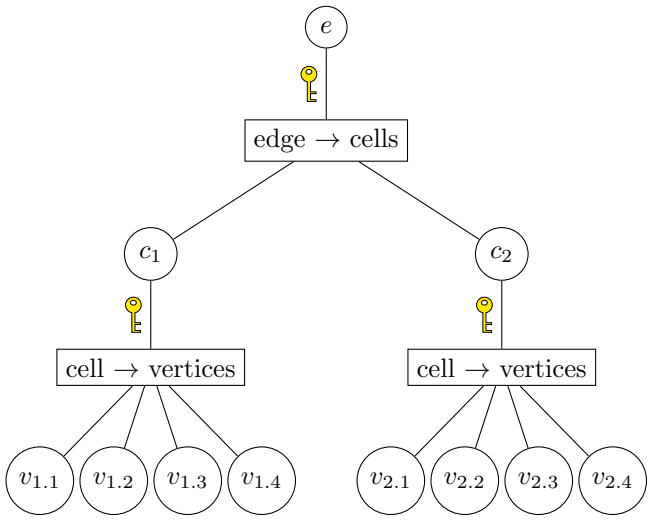


Figure 2.8: In this example, we consider a core-computation operating over the edge set, with e being the indexing variable into the edge set. The edge → cells map is indexed by e to obtain the two cells c_1 and c_2 incident on the edge e. The cell → vertices map is then indexed by both c_1 and c_2 to obtain their respective vertices. The key symbol denotes indexing into the map below, using the indexing variable above.

As we define it, however, kernel functions are in fact more general than a stencil computation, as they access neighbouring elements across different operating sets.

The kernel function is applied to the operating set elements in no particular order; the indexing variables, however, are passed to the kernel in some known order, typically in the order stored in the relation-map.

2.3.4 Expected operation

Given all the above, a core-computation is then performed as follows:

1. Iterate over the elements of the operating set, in no particular order.

2. For each element iterated over:

   (a) Gather any indexing variables as defined by the relation-map tree. This may involve indexing variables obtained through a chain of relation-maps.

   (b) Call the kernel function, passing the gathered indexing variables in some known order. The kernel function may access any associated data using these indexing variables.

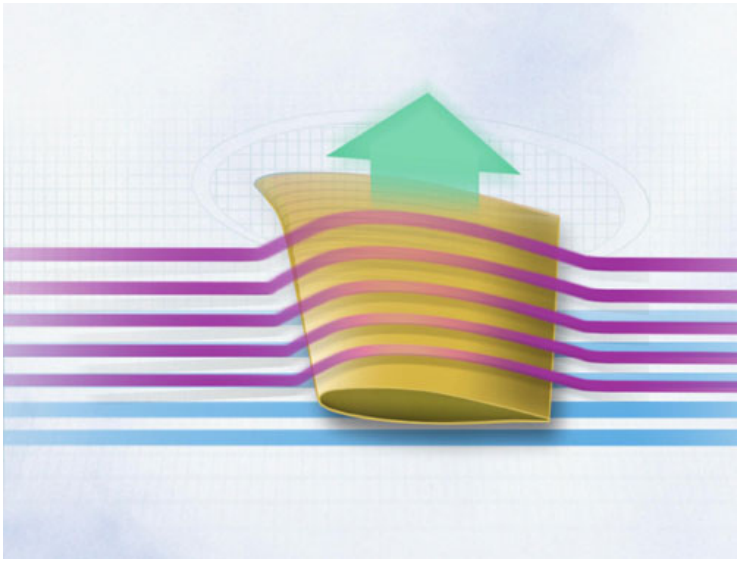


Figure 2.9: The cross-section used to obtain the airfoil shape. Incoming air flow is split between the upper surface (purple) and the lower surface (blue). The image was obtained from [3].

2.4 Background on airfoils

While not strictly needed for understanding our work, we nonetheless describe briefly airfoils and their function to offer a broader context. Much of this section was adapted from [1], [16] and [3]. Our description is nonetheless undoubtedly an overly simplistic one, and we would recommend that the aforementioned literature be sought for a fuller picture.

An airplane achieves flight by creating a lower air pressure over the wing (*the upper surface*) whilst maintaining a higher air pressure below the wing (*the lower surface*). The exact way in which this is achieved is characteristic of the wing shape as well as other factors. The pressure differential causes air in the lower surface to push towards the upper surface, creating a lift force. If the lift force is sufficient to counteract the gravitational force, the airplane flies.

An airfoil is the two-dimensional cross-section *shape* of a wing. They are used to model the hydrodynamics (fluid motion) surrounding a particular wing shape in different contexts, including the velocity and angle of motion (known as the *angle of attack*). Figure 2.9 shows how the cross-section is taken, as well as the modelled air flow.

In 1929, the National Advisory Committee for Aeronautics (NACA) began to study various airfoils. They developed families of airfoil constructions parameterized by various geometric variables, depicted in figure 2.10. We use specific instantiations of these airfoil families as benchmarks for Crystal.

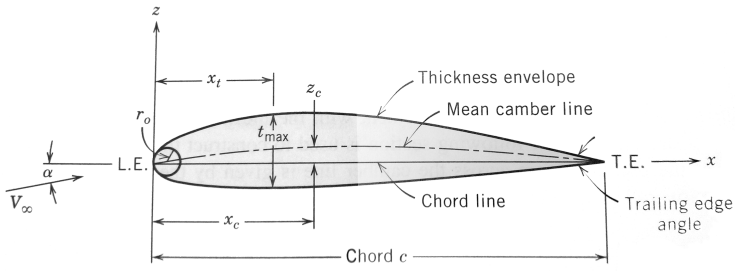


Figure 2.10: A depiction of the geometric properties of airfoil. The image was obtained from [16].

2.5 Chapter summary

In this chapter we gave a basic description of a mesh as a mathematical model. In addition we defined the concept of the mesh a data structure and how it is used in computation. Finally, we explained the significance of the airfoil in order to have a framing context for our running example.

Chapter 3

Related works

We present a selection of prior works which touch upon the area of mesh structure, or are related to it in appealing ways.

3.1 Structure detection and extraction

Of all the works discussed in this chapter, [27] by Tautges is the most similar. The paper discusses the merits of structured meshes in great depth. It also defines the concept of disparate structured regions in a mesh, and discusses strategies for fusing them into a hierarchy of structures. A structure construction algorithm is also discussed very briefly.

Adapting the mesh model to use a structured representation for the benefit of the relevant application domain is common practice. Rocca et al. [24] address the particular problem of neighbourhood search: finding the set of vertices within a certain distance from a given vertex. They define a spatial index, a hierarchy of vertex clusters, that yields good approximations to neighbourhood search queries.

The problem of memory access indirections, associated with representing the mesh topology, is cited as a key motivation for basing their mesh data structure (the aforementioned spatial index) on a geometric model which does not require storing topological information. This is in sharp contrast to our approach of transforming the mesh topology representation, and treating the mesh geometry as arbitrary associated data.

Eppstein et al. [10] describe a method for partitioning quadrilateral meshes into structured regions using the *motorcycle graph* construction, whose inspiration came from the 1982 movie Tron. The algorithm works by placing particles on each *extraordinary* vertex[1] in the mesh, and advancing them along edges until they collide. The enclosed regions formed by the paths represent structured regions.

[1] This is in contrast to an *ordinary* vertex, which the authors define as "a non-boundary vertex incident with four edges or a boundary vertex incident with at most three edges."

Whilst the motivations are different, with the emphasis on applications in mesh compression and detecting mesh isomorphisms, the approach may be applicable to the scientific computation domain. Indeed, the authors state this explicitly:

> These techniques may apply to areas beyond graphics such as scientific computation. Code for the finite element method can be greatly streamlined when applied to structured quadrilateral meshes. By partitioning unstructured meshes into structured sub-meshes, it should be possible to achieve similar speedups for semi-regular meshes.

We note, however, that the method makes a stronger assumption about the mesh, in that its "structure is interrupted by a *small number* of extraordinary vertices that do not have degree four [emphasis added]".

Makem et al. [18] utilise properties of the modelled object to generate meshes with structured regions inherent. The presented method detects long thin shapes with simply defined geometry, such as length and curvature, and generates an appropriate structured region representing these shapes.

3.2 Structure in parallel computation

There is a plethora of work on mesh partitioning optimised for parallel computation. The techniques presented often frame their objectives around these maxims:

1. Maximize intra-partition locality, thereby minimizing cross-partition communication.
2. Minimize partition size, subject to it being sufficiently large to counterbalance the communication overheads.
3. The partitions should be balanced and plentiful in number, so as to utilise parallelism. "A parallel computation is often only as efficient as the evenness with which its workload is distributed over the processors in a parallel machine."

Objective 1 is certainly a desirable property for our structured regions, and is in fact precisely what we set forth to perform. On the other hand, objectives 2 and 3 are rather misaligned with our needs; indeed, a monolithic structured region spanning the entire mesh would represent the best case for us.

Thus some of the techniques which hold these conflicting[2] objectives may not be fully compatible, though they undoubtedly offer helpful inspiration.

With that said, extending our techniques towards parallel computation is an obvious future step, and such future works should likely re-evaluate their objectives in lieu of this.

[2]The objectives are in conflict in our context. These objectives are more suited for their parallel context, naturally

A technical report by Ridley [23] briefly discusses methods for partitioning a mesh so as to maximize spatial data locality, in other words adjacent elements tend to have memory locations that are close. The partitioning is applied by recursively bisecting the mesh geometrically, such that geometrically close points tend to cluster together. The emphasis of the report is towards improving the performance of parallel computation, but the benefits of partitioning extend to serial processing as well.

Li et al. [17] follow a similar theme of parallel computation, although their methods deal with adaptive mesh refinement, where a mesh is dynamically refined in regions with a high calculation error. The refinement process induces a hierarchical structure, which the presented partitioning algorithms aim to exploit.

Bergen et al. [2] employ structure-aware mesh refinement techniques, which construct a hierarchy of structured regions with each iterative refinement to the mesh. It is suggested that different element types, such as edges and faces, refined and stored separately, such that their distinct structure can be represented.

Since we use a simple structure representation, we make use of augmented structured regions, with different elements' structured region represented in a hierarchy.

Reed et al. [22] focus on obtaining partitions best-suited to the *stencil structure* associated with a particular computation, that is its neighbour-access pattern. They derive partition shapes for some common stencil structures, optimised to minimise inter-partition communication costs.

Tang et al. [26] present a full-fledged *Pochoir Stencil Compiler*. It specifies a domain-specific language that allows users to write a higher-level specification of a stencil computation embedded in C++ code. The compiler then automatically generates very efficient *cache-oblivious*[3] parallel loops that execute the stencil computation.

As discussed in subsection 2.3.3, this is similar to our approach, where the structured regions are detected on the basis of relation-maps, such as cell-vertex or edge-cell maps.

3.3 Exploiting structure

Eppstein et al. [9] explore the problem of approximate topological matching between given quadrilateral meshes, that is detecting isomorphisms between their submeshes. To this end they discuss various techniques based on *particle shooting*[4]: "particles" are placed on certain vertices (for example, extraordinary vertices[1]) and are then "'fired" along the mesh using certain rules.

[3]The authors of [12] define a cache oblivious algorithm as that which "[does not contain] parameters (set at either compile-time or runtime that can be tuned to optimize the cache complexity for the particular cache size and line length"

[4]The motorcycle graph construction discussed in the paper by the same authors [10] is based on particle shooting.

One such algorithm, termed "The Greedy Algorithm" by the authors, is shown to suffer from a problem when particles advancing along the same front can get out of sync. The problem is remarkably similar to our discussion about contiguous detection in subsection 5.2.2.

3.4 Chapter summary

In this chapter we discussed the similarities and differences between our work and that of a variety of published works centred at or homing around our scope of interest. We saw that the inefficiencies associated with unstructured representation are a common pain point, and we saw the great interest in the area of detecting structure to aid parallel computations. Some works presented exploited structure for other intriguing purposes.

Chapter 4

Diving into the problem

We begin by walking through an example mesh and examining the properties of the structure found within.

4.1 A basic definition of structure

What we would like is a form of structure which is a) representable as a data structure, and b) efficient in terms of performance.

Given our semantic knowledge about the mesh model we can ascertain some facts about relation-maps:

- They are sparse: element arity is very small compared to the number of elements, and is in fact unrelated to it.
- They have a high clustering coefficient: Relationships tend to be localized, forming tightly connected clusters.

These properties arise as a consequence of meshes modelling real-world phenomena that exist in a Cartesian space. On this basis, we consider a spatial embodiment of element relationships, organising the elements in a discrete space such that the uniform relationship is apparent.

Bear in mind that this approach carries no relation to any geometric data associated with elements, such as the coordinates of vertices. To make this distinction clear, as well as to emphasize its discrete nature, we address this Cartesian-like space by rows and columns rather than x and y coordinates.

4.1.1 Example: Naca0012 mesh

Figure 4.1 shows a small extract from the NACA0012 mesh[1], showing the cross section of an airfoil mesh and its interaction with surrounding fluid. The mesh is discretized into quadrilateral cells over which computations are performed.

[1]Thanks to Dr. Peter Vincent, George Ntemos and Harry Davis

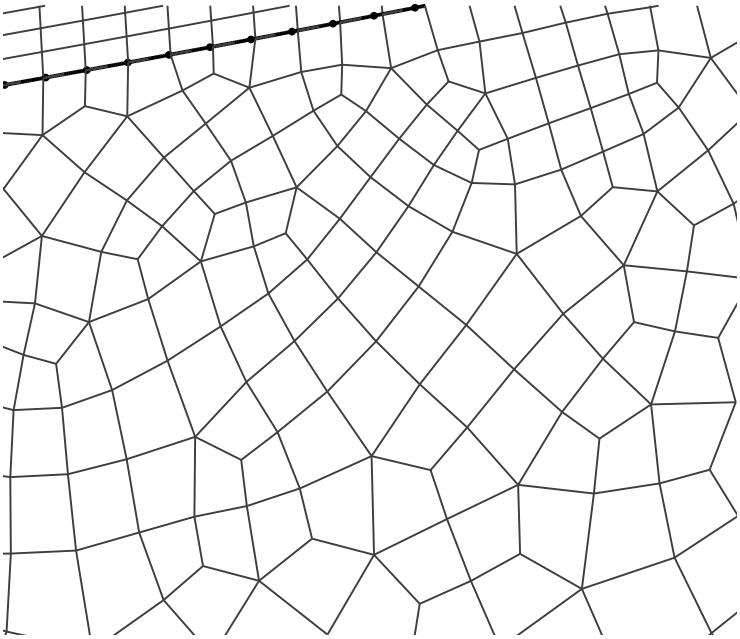


Figure 4.1: Extract of the NACA0012 mesh.

The vertices in the highlighted region of figure 4.2 seem like good candidates for a *"structured region"*, forming a two-dimensional lattice in a discrete Cartesian space. This *"structured region"* has the properties outlined below.

4.1.2 Desired properties of a structured region

1. All vertices have a uniform arity of four.

2. Every vertex has a consistent discrete direction (for example the cardinal directions: north east, south, west) with respect to the other vertices. In other words, the direction is transitive: if vertex a is above vertex b, and vertex b is above vertex c, then vertex a is above vertex c. For a non-example see figure 4.3.

We can propagate this inherent structure from the mesh model to the underlying data structure, representing this two-dimensional lattice using a two-dimensional array. Vertices may be assigned Cartesian coordinates, but in spirit of the space's discreteness we shall use rows and columns instead. See figure 4.4.

Let us call V the set of all vertices, and $V_{str} \subseteq V$ the set of vertices in the *"structured region"*.

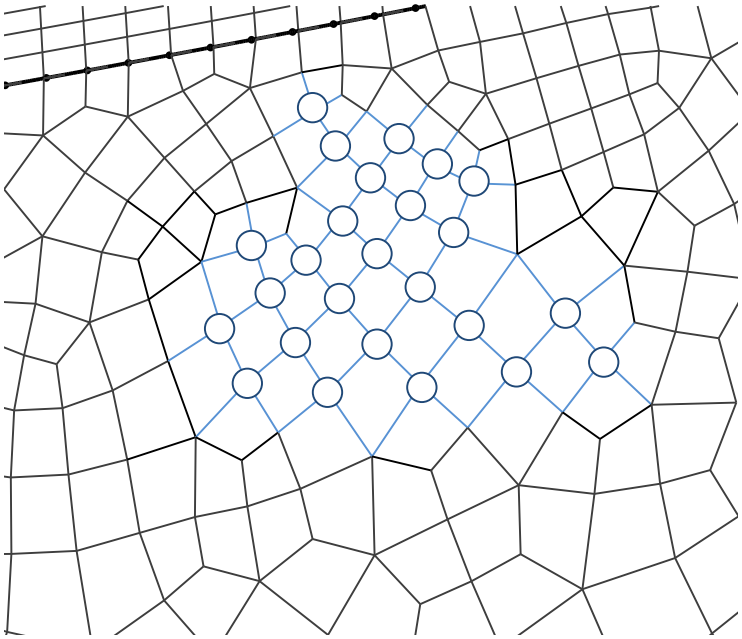


Figure 4.2: Highlighted vertices which exhibit a form of *"structured region"*.

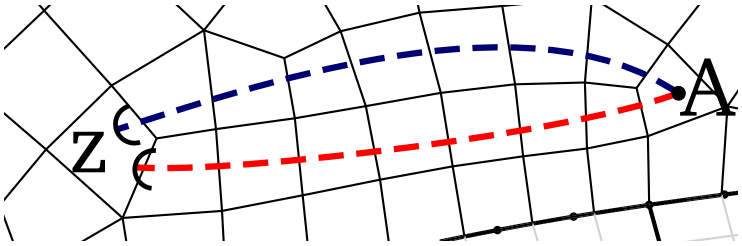


Figure 4.3: An example of inconsistent direction. We can traverse cells in *"one direction"* by following the edge parallel to the one we entered from. If we start from A and traverse the cells in one direction (the blue path) we reach Z. If we start from A and traverse the cells in an orthogonal direction (the red path) to the first path, we also reach Z! Is Z then *"above"* A or *"to the side of it"*?

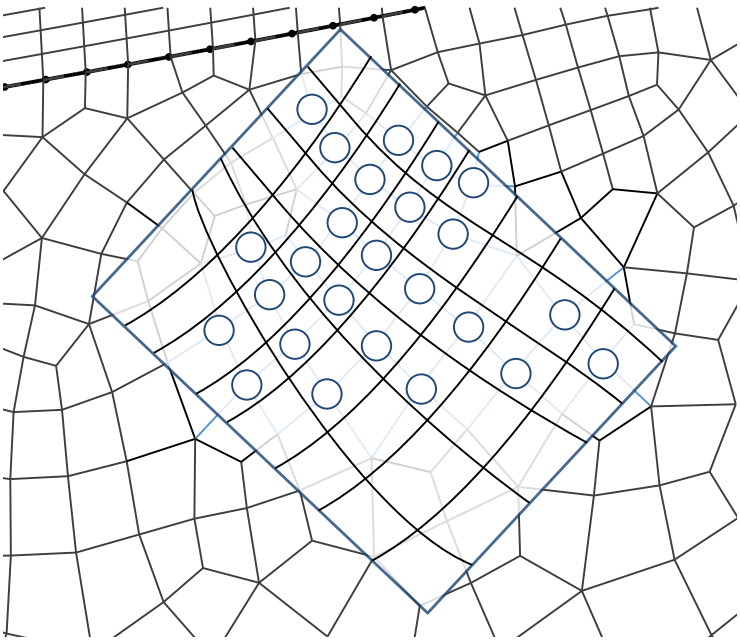


Figure 4.4: The overlaid grid shows the lattice structure more clearly.

4.1.3 Representing the vertex-vertex adjacency

Now consider the vertex-vertex adjacency relation $Adj_{VV} : V \mapsto V$ in context of the *"structured region"* V_{str}. We can directly locate a particular neighbour of any vertex, for example its north neighbour, so long as that neighbour is also within the structured region. This restricts the set of vertices with fully-accessible neighbours to those which are not on the borders or the fringe of the *"structured region"*. This is the subset of vertices $V_{adjstr} \subseteq V_{str}$ which are structured *with respect to* Adj_{VV}. This induces a new relation which operates purely within the structured region:

$$Adj_{V_{adjstr}V_{str}} : V_{adjstr} \mapsto V_{str}$$

Figures 4.5 and 4.6 illustrate these two sets.

The key insight we make is that for structured regions in a mesh we need not represent set relationship maps explicitly; the uniformity of set relations allows us to deduce the relationships. We can encode the relation $Adj_{V_{adjstr}V_{str}}$ very simply. Given a vertex $n_{r,c} \in V_{adjstr}$, located at row r and column c, its four vertex neighbours are $n_{r,c-1}$, $n_{r,c+1}$, $n_{r-1,c}$, and $n_{r+1,c}$. This is illustrated in figure 4.7.

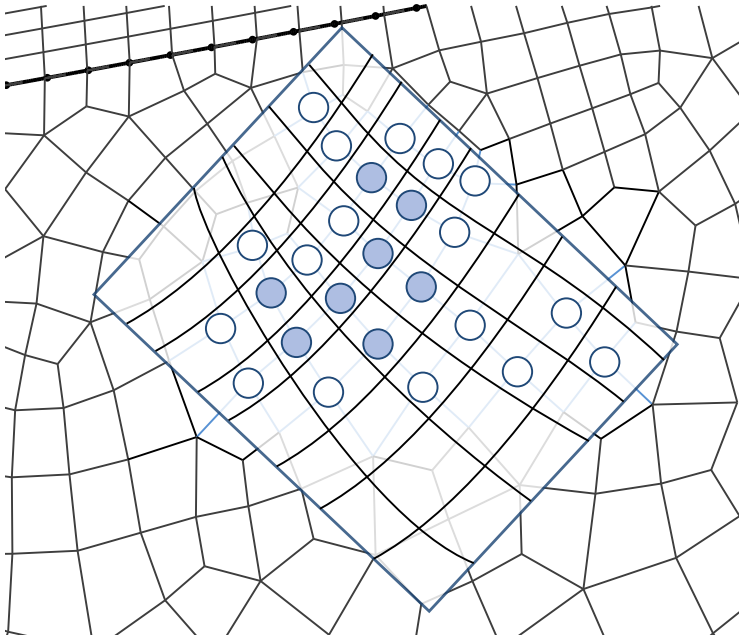


Figure 4.5: The interior structured vertices (those not on the fringe) are highlighted in dark blue.

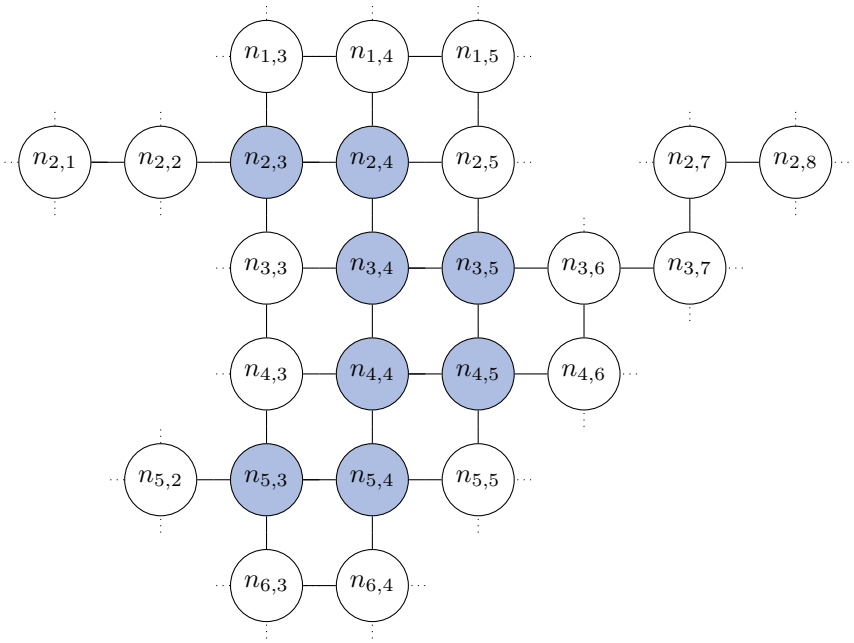


Figure 4.6: The vertices in figure 4.5 represented as a graph.

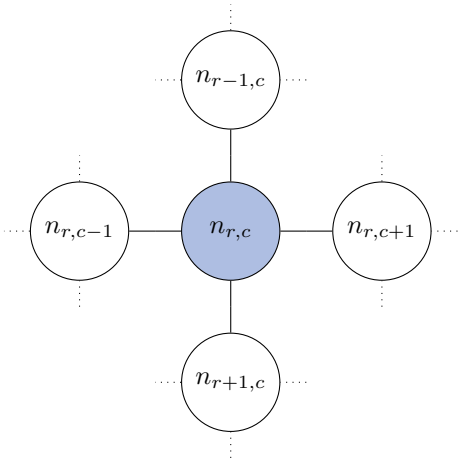


Figure 4.7: The neighbours of a structured vertex.

4.2 Mesh structure as a data structure

The choice of data structure to represent the structured region is a key one, touching on various aspects:

- Scope of structure representation: what level of structure can be represented.
- Implementation complexity of structure detection: how complex a detection algorithm is required.
- Runtime performance of structure detection: the time and space complexity of the detection algorithm.
- Storage requirements for detected structure.
- Runtime performance of computation over the structured region.

We analyse several possible data structure representations.

4.2.1 Structure bit-mask

Represent the bounding box around the structured region as a two-dimensional bit-mask, with each bit representing a vertex position in the superimposed grid. An *on* bit indicates that the corresponding vertex is part of the structured region, an *off* bit otherwise.

A bit-mask can represent *any* structured region which adheres to the properties listed in section 4.1.2. They are very flexible, capable of representing complex formations as well as handling small anomalies in structure.

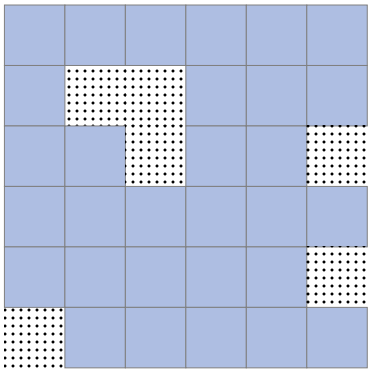


Figure 4.8: An example of a structured region which can be bit-mask defined. The dotted regions indicate unstructured cells. The blue cells are structured cells. Its efficiency ϵ is $\frac{30}{36} \approx 83\%$.

Let us define the *efficiency* ϵ of a bit-mask as the percentage of its bits which are *on*, as the *off* bits represent wasted vertices. Bit-masks with high ϵ are favourable for several reasons:

- *Off* bits increase the storage space of the structured region, up to $O(|V|)$ in the worst case, as opposed to $O(|V_{str}|)$.
- *Off* bits similarly worsen the runtime performance, again up to $O(|V|)$ due to wasted execution.

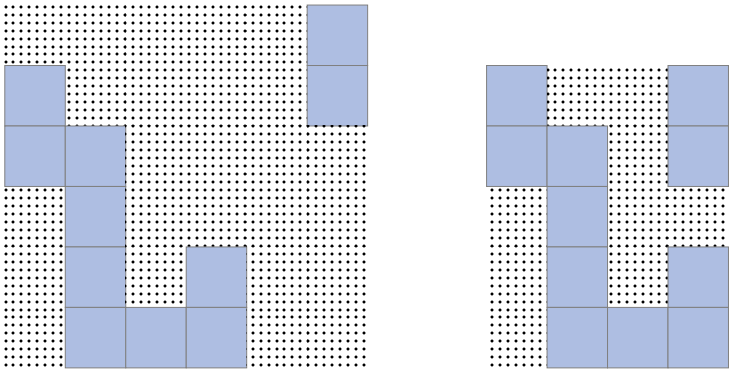
Figure 4.8 shows an example of a good bit-mask candidate.

A detection algorithm would likely be implemented using a variant of a general graph search algorithm such as breadth-first search or depth-first search. Further consolidation work may be needed to compact disconnected structured components in order to maximize ϵ. The potential benefits of compaction are illustrated in figure 4.9.

4.2.2 Row-specific boundaries

Represent the structured region as a sequence of consecutive fixed-length rows, permitting vertices outside the structured region to cluster at the beginning and ends of each row. A structured region is then represented by the row-length, as well as individual begin and end offsets for each row. This is exemplified in figure 4.10.

Storing row-specific boundaries reduces the storage requirement by the order of row-length times. No executions are wasted as the loop iterations are constrained to vertices in the structured region. A detection algorithm can be implemented as a simple row-by-row traversal in linear time and space.



(a) Originally detected structured components.

(b) Compacted structured components.

Figure 4.9: A demonstration of the benefits of disconnected structured components compaction. The detected structured cell components in 4.9a are unnecessarily sparse and occupy more space than necessary. After compaction in 4.9b the space is used is reduced by over 40%.

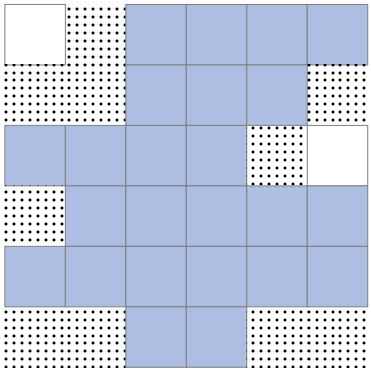


Figure 4.10: An example of a structured region which can be represented using row-specific boundaries. Note the white cells, which represent unused structured regions near the boundaries, which cannot be included as part of the structured region due to the constraints imposed.

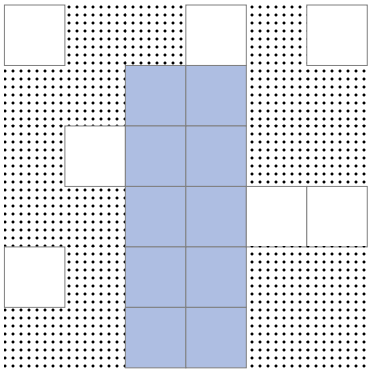


Figure 4.11: An example of a rectangular structured region. Note the abundance of unused structured cells.

On the downside, this scheme trades some of the flexibility offered by the bit-mask representation, in particularly anomalies in structure which do not occur at the boundaries.

4.2.3 Full rectangle

Represent the structured region as a rectangle of given length and width, with all elements corresponding strictly to vertices within the structured region. The structured region is thus simply represented by the length and the width: the number of rows and the number of columns. Figure 4.11 shows an example of a rectangle-representable structured region.

The storage requirement is now constant with respect to the size of the structured region. Loop iterations are fixed for each region, enabling further optimisation opportunities. A detection algorithm, as above, can be implemented as a simple row-by-row traversal.

On the downside, the scope of structure representable is limited, being only capable of representing rectangles proper. Nonetheless, we stick with detecting rectangular structured regions for the remainder of this work as it seems to be the most promising choice.

4.3 Chapter summary

In this chapter we derived intuitive definitions structure in a mesh in light of an excerpt from an airfoil mesh. We also discussed different representations in which they can be manifested, and the advantages and disadvantages of each.

Chapter 5

Structure growth algorithms

Given a working definition of structure in a mesh, we build on this definition and introduce structure detection algorithms.

5.1 Key concepts

An *absolute structured position* refers to the position of a structured element with respect to the boundaries of its structured region. A *relative structured position* refers to the position of a structured element with respect to other structured elements within its structured region. Figure 5.1 shows this through an example.

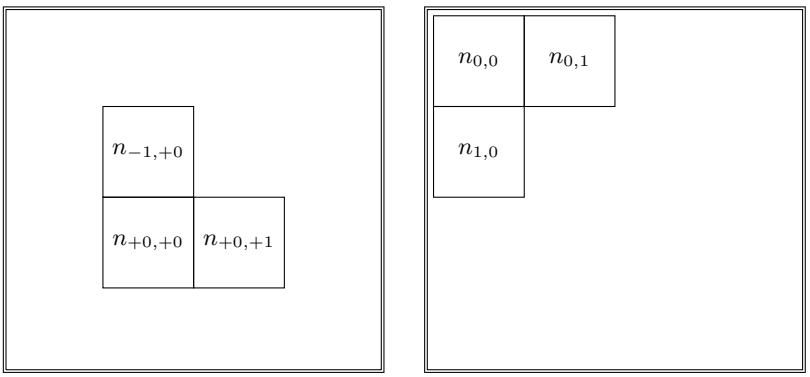
5.2 Properties of detection algorithms

5.2.1 Eager detection

An eager (or greedy) detection algorithm will include every structured element it finds immediately, regardless of the long term consequences. While this strategy may lead to suboptimal results, it avoids backtracking the structure detection which may be prohibitively costly. Figure 5.2 depicts the sub-optimality of eager detection in contrast with non-eager detection.

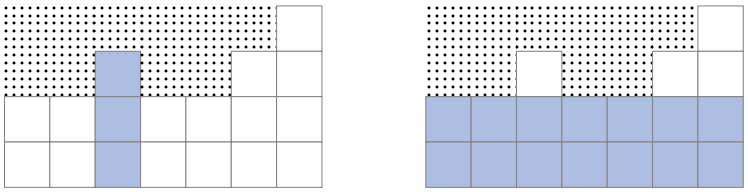
5.2.2 Contiguous detection

An algorithm which exhibits continuous detection always adds a structured element which is contiguous to the structured region thus far. The implication is that the relative structured position is always known. This greatly simplifies detection, as all adjacent structured elements are known at any point in time,



(a) Relative structured position **(b)** Absolute structured position

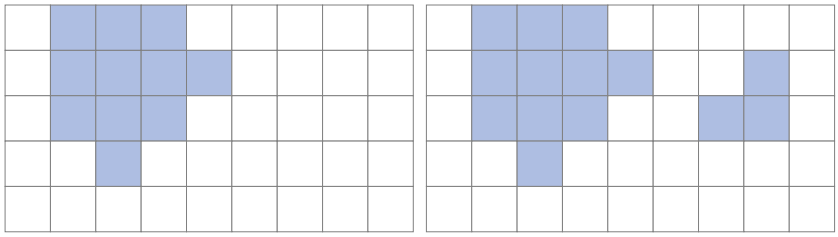
Figure 5.1: Depiction of *relative* structured position versus *absolute* structured position. The row and column counts are increasing down and to the right, respectively. The borders indicate structured regions, whose origin resides in the top-left corner.



(a) Eager detection may greedily add the northern cell, yielding a suboptimal structured region.

(b) A non-eager algorithm could instead decide to ignore the northern cell, yielding a larger structured region.

Figure 5.2: Eager versus non-eager detection. The dotted region represents unstructured elements. Blue cells denote structured elements detected as forming a rectangular structured region; white cells denote the remaining structured elements.



(a) Contiguous detection always adds cells adjacent to the structured region detected thus far.

(b) Non-contiguous detection may add cells which do not border the structured region detected thus far.

Figure 5.3: Contiguous detection versus non-contiguous detection algorithms. White cells denote structured elements which have not been added to the structured region. Blue cells denote structured elements detected thus far.

and structured elements need not be repositioned in the structured region. Figure 5.3 contrasts contiguous detection with non-contiguous detection.

In the case of non-contiguous detection, any non-contiguous blobs need to be consolidated. These blobs may be one of three cases:

1. Disjoint:

   These may simply be taken as two separate structured regions.

2. Compatible:

   The blobs can be merged in a lossless manner to form a single structured region.

3. Incompatible:

   The blobs cannot be merged without loss of structure due to inconsistencies between the blobs. It is then necessary to discard some structured elements.

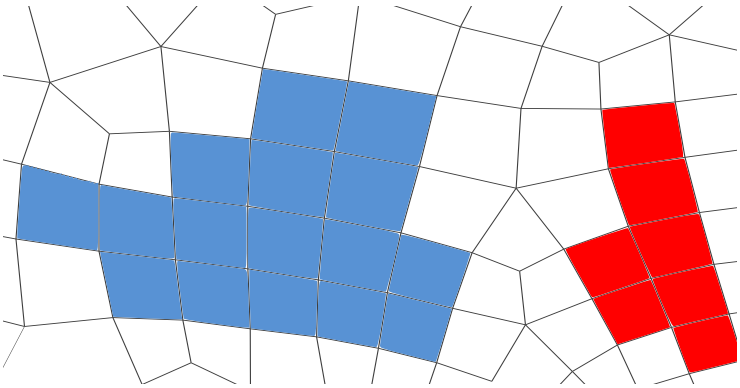
Examples of the three cases are shown in figure 5.4.

5.2.3 Post processing requirements

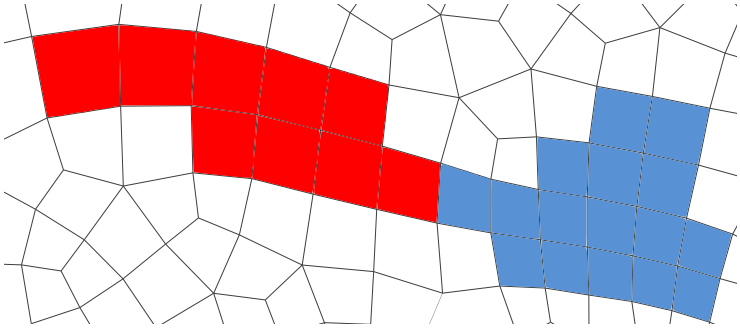
Different algorithms will require different levels of post-processing in order to yield a rectangular structured region. Some may require a simple operation, such as trimming incomplete rows, while others may require more complex operations to achieve this goal.

5.2.4 Detection traversal patterns

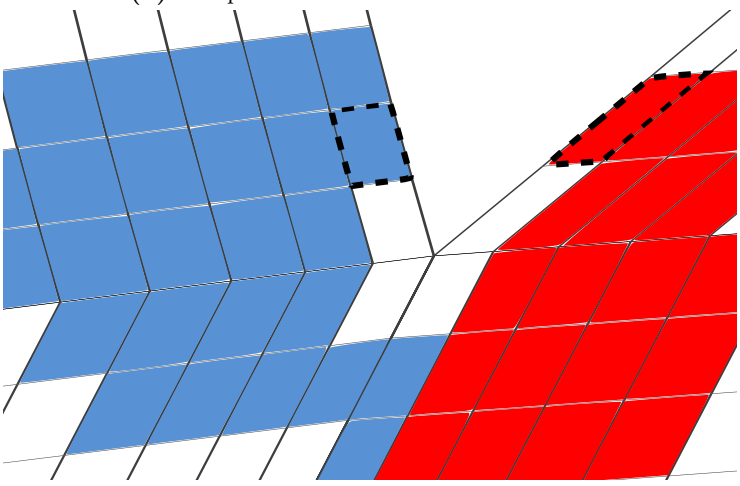
The order in which structured elements are detected in a structured region is important; it imposes some constraints on the data structure representing it.



(a) Disjoint blobs of structured elements.



(b) Compatible blobs of structured elements.



(c) Incompatible blobs of structured elements. The two dashed cells are not adjacent in the mesh, but if added as structured element they would have adjacent positions in the structured region.

Figure 5.4: Cases that may arise with non-contiguous detection.

Given the dimensions of the structured region, and knowledge of the absolute structured positions of elements as they are discovered, a simple 2D array allocation would suffice. Any detection order, as is convenient, may be used in this case. However, neither of those facts are known a priori in general.

Various detection traversals orders and their merits are discussed below. Figure 5.6 outlines some examples.

Single-row append-only

The structured region is grown in a constant direction, for example a single row of structured elements, appended to consecutively. This can be implemented efficiently using either a singly-linked list or a dynamic array with amortized constant time append operation.

Single-row append/prepend

The structured region is grown in either of two directions, for example a single row of structured elements, appended and prepended to. This can be implemented efficiently using a double-ended queue with amortized constant time append and prepend operations.

Row-oriented detection

The structured region is represented as a group of rows, with the elements in individual rows grown using one of the above methods. The order in which the rows themselves are grown may also utilize the same methods, with a nested data structure being a suitable implementation. For example, if rows are detected in an append-only fashion, and the individual elements are detected using append and prepend operations, then a suitable data structure would be a singly-linked list of double-ended queues.

Indeterminate order detection

The structured region is grown in a non-linear order: grown elements may not always be contiguous to the structured region thus far. If the growth is indeed non-contiguous, the relative structured positions are *not* always known, and structured elements may need to be repositioned. A possible implementation would be a jagged 2D array, that is an array of arrays, which is expanded as needed. A flat-array-based 2D array would (in the worst case) require reallocating all elements upon expansion, as opposed to reallocating a single row in the case of a jagged 2D array.

Assuming contiguous structure detection, then when adding a new structured element e_{new} the following holds: a) we know exactly which existing structured elements must be adjacent to e_{new}, and b) we only need to check the adjacency between e_{new} and existing structured elements. On the other hand if structure detection is non-contiguous, then we may later discover that some

structured elements are no longer mutually compatible. We thus must check the adjacency between existing structured elements as we add new ones. Figure 5.5

5.3 Structure growth algorithms

We now briefly sketch some structure growth algorithms which grow a rectangular structured region.

5.3.1 Length-first search

1. Starting from a seed vertex, grow a quad.
2. The quad is grown along one axis, both forwards and backwards, as far as possible, forming the length of the structured region. This is a per-row append/prepend traversal.
3. The row grown above is extended along the orthogonal axis, both forwards and backwards, as far as possible. Each expanded row must have the length of the initial row exactly, forming the width of the structured region. This is a row-oriented append/prepend traversal.

The detection is clearly contiguous, with all structured element insertions running in amortized constant time. Only a constant amount of extra storage is required.

The algorithm is also an eager one, deciding the length of the structured region based on the first row it detects. This simple approach, however, can result in suboptimal detection.

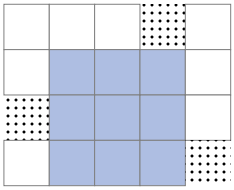
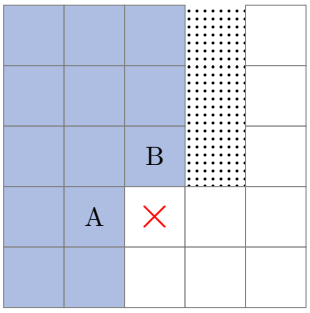


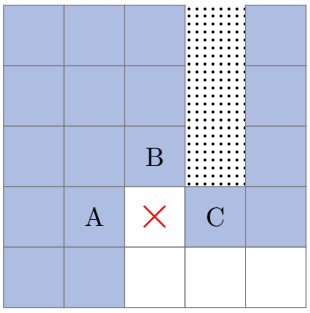
Figure 5.7: Example run of the length-first search algorithm.

5.3.2 Descending-staircase search

1. Follow the same steps as in length-first search, with one exception: in step 3, each expanded row may have a length which is at most the length of its predecessor.
2. Find the rectangle with the maximum area. Daniels et al. discuss in [6] algorithms for obtaining such a rectangle, which has time complexity $O(\alpha(n))$, where$\alpha(n)$ is the slowly growing Ackermann function.

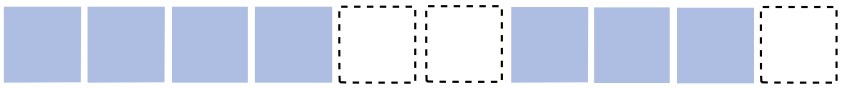


(a) Contiguous detection, with $\times$ denoting the currently considered element. We must check that $\times$ is adjacent to both A and B, but is not adjacent to any other structured element detected thus far.



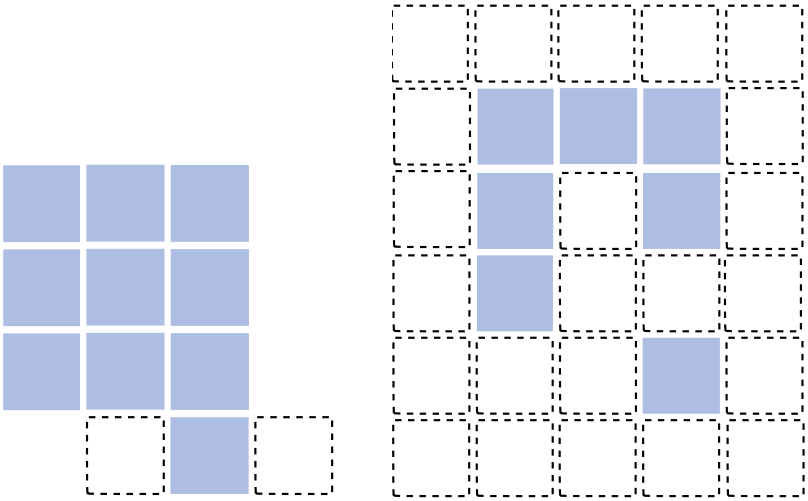
(b) Non-contiguous detection, with $\times$ denoting the currently considered element, which if added will join two disjoint components. We must check that $\times$ is adjacent to A, B, and C, but not adjacent to any other existing structured element. Furthermore, we must check that the existing structured elements in one component are *not* adjacent to any structured elements in the other component, except through $\times$.

Figure 5.5: Comparing the costs of contiguous structure detection 5.5a and non-contiguous structure detection 5.5b.



(a) Single-row append detection.

(b) Single-row append/prepend detection.



(c) Row-oriented detection, with rows detected in an append-only fashion, and elements within rows in an append/prepend fashion.

(d) Indeterminate order detection. Note that some possible regions of expansions are not adjacent to the presently detected structured region.

Figure 5.6: Examples of detection traversal patterns. The blue blocks represent presently detected structured regions. The dashed pink blocks represent possible regions for expansion.

The algorithm has the same property of being contiguous, and it runs under the same time and space complexity requirements.

Unlike length-first-search, however, this algorithm is not an eager one, as it defers the decision of selecting which elements belong in the structured region.

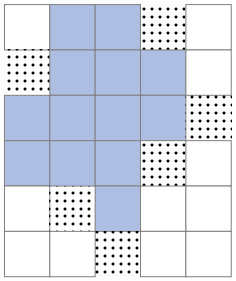


Figure 5.8: Example run of the descending-staircase search algorithm. The starting point is in the upper-middle cell.

5.3.3 Scan-line search

This algorithm is identical to descending-staircase search, except that there is no restriction on the length of each expanded row. It has virtually the same properties of descending-staircase search, except:

- The complexity of finding the rectangle with the maximum area increases to $O(nlog^2n)$.

- By constraining its row expansion, descending-staircase search avoids taking long thin routes which are heuristically less likely to lead to a useful structure.

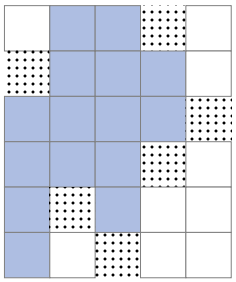


Figure 5.9: Example run of the scan-line search algorithm. The starting point is in the upper-middle cell.

5.4 Detecting multiple structure regions

Using our structure-growth algorithm of choice, we can grow multiple *disjoint* structured regions by the following algorithm:

1. Let $\Xi = \varnothing$ be the set of detected structured regions.
2. Let $V_{visited} = \varnothing$ be the set of visited vertices, and $V \setminus V_{visited}$ the set of unvisited vertices.
3. Choose a random unvisited vertex $v \in V \setminus V_{visited}$. If none exist, abort.
4. Add v to $V_{visited}$.
5. Apply a structure-growth algorithm starting from v.
6. If the structure-growth fails, go back to step 3.
7. Otherwise, add the detected structured region to Ξ, and all the vertices in the structured region to $V_{visited}$.
8. Go back to step 3

5.5 Chapter summary

This chapter covered algorithms for the detection of structured regions. An abstract discussion of their various properties was presented, and the desirable traits brought forth by each. Then, a selection of concrete algorithms were described, and their merits and weaknesses examined. Finally, we describe a general method for detecting multiple structured regions.

Chapter 6

The length-first search algorithm

Taking forth our length-first search algorithm, we expand fill in the details glossed over by our earlier discussion, delineating it into a formal algorithmic specification.

6.1 Inputs

1. A non-reflexive and symmetric vertex-vertex adjacency relation:
$$Adj_{VV} : V \mapsto \mathcal{P}(V)$$
2. A set of visited vertices
$$V_{visited} \subseteq V$$
3. An unvisited start vertex
$$v_{init} \in V \setminus V_{visited}$$

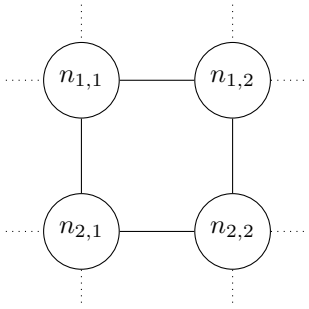
6.2 Outputs

1. A structured set of vertices $V_{structured} \subseteq V \setminus V_{visited}$ forming the extracted structured region. The vertices $V_{structured}$ are structured on a 2-dimensional Cartesian lattice with m rows and n columns.

6.3 Phase 1: Grow a quad

Starting from the initial vertex v_{init}, call it $n_{1,1}$, we would like to discover three other vertices $n_{1,2}$, $n_{2,1}$, and $n_{2,2}$ such that $\begin{bmatrix} n_{1,1} & n_{1,2} \\ n_{2,1} & n_{2,2} \end{bmatrix}$ forms a valid quad in a structured quad region. They must satisfy the following constraints:

- Each of the four vertices must have exactly 4 neighbours.
- Each of the following pairs of vertices are neighbours: $n_{1,1}$ and $n_{1,2}$; $n_{1,2}$ and $n_{2,2}$; $n_{2,2}$ and $n_{2,1}$; $n_{2,1}$ and $n_{1,1}$.
- Each of the vertices must *not* neighbour any vertex that has been visited thus far, apart from those vertices explicitly mentioned.

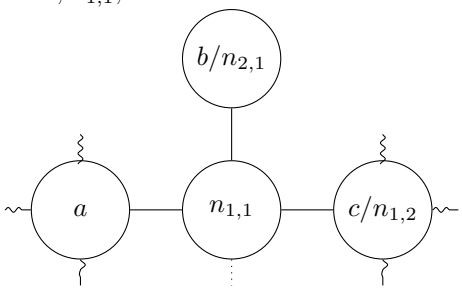


6.3.1 Algorithm

1. If v_{init} does not have exactly 4 neighbours, that is $|Adj_{VV}(v_{init})| \neq 4$, return immediately with $V_{structured} = \varnothing$.
2. Let $n_{1,1} = v_{init}$, and let $a, b, c \in Adj_{VV}(v_{init})$ be distinct vertex neighbours of v_{init}. Consider vertices a and b. If they do not both have exactly 4 neighbours, then they cannot form a part of a structured quad region. Return immediately with $V_{structured} = \varnothing$.
3. Otherwise, there are three cases:
   (a) a and b have exactly one neighbour in common, which must be $n_{1,1}$ by construction, expressed by:

   $$|Adj_{VV}(a) \cap Adj_{VV}(b)| = 1$$

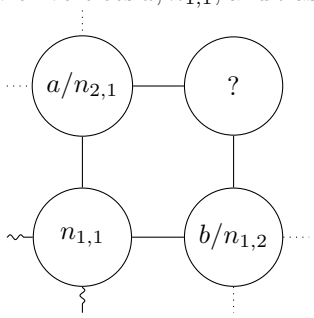
   $a, n_{1,1}, b$ are *topologically* along a straight line of a structured grid, and hence cannot form a quad. We therefore consider b, $n_{1,1}$, and c instead as candidates, letting $n_{2,1} = b$ and $n_{1,2} = c$.



(b) a and b have exactly two neighbours in common, one of which must be $n_{1,1}$ by construction, expressed by:

$$|Adj_{VV}(a) \cap Adj_{VV}(b)| = 2$$

We continue with vertices a, $n_{1,1}$, and b as candidates, letting $n_{2,1} = a$ and $n_{1,2} = b$.



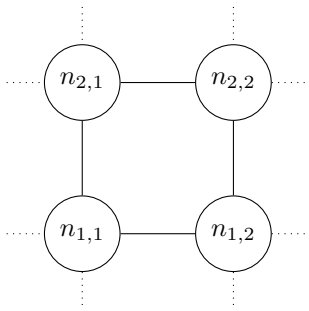
(c) a and b have more than two neighbours in common[1], expressed by:

$$|Adj_{VV}(a) \cap Adj_{VV}(b)| > 2$$

We cannot form a quad, and hence return immediately with $V_{structured} = \varnothing$.

4. Find the common neighbours of $n_{2,1}$ and $n_{1,2}$. If these are not exactly two neighbours, return immediately with $V_{structured} = \varnothing$.
5. One of the two neighbours must be $n_{1,1}$ by construction. Let the other neighbour be $n_{2,2}$.
6. Let $N = \{n_{1,1}, n_{1,2}, n_{2,1}, n_{2,2}\}$. If any vertex $n \in N$ is in $V_{visited}$, that is $N \cap V_{visited} \neq \varnothing$, then return immediately with $V_{structured} = \varnothing$. Otherwise add the vertices in N to $V_{visited}$.
7. Ensure for every vertex $n \in N$ that its visited neighbours, $Adj_{VV}(n) \cap V_{visited}$, are exactly those explicitly stated above. If this is not the case, remove N from $V_{visited}$ and return immediately with $V_{structured} = \varnothing$.
8. Set $V_{structured} = \begin{bmatrix} n_{1,1} & n_{1,2} \\ n_{2,1} & n_{2,2} \end{bmatrix}$, and continue to the next phase.

The structured region looks as follows thus far.

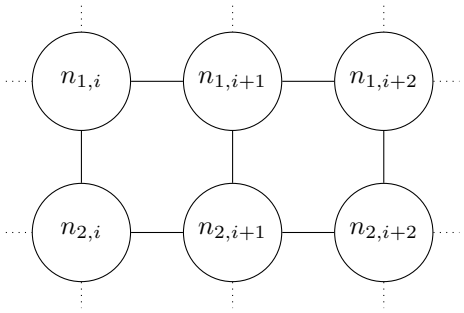


[1] Note that the set is non-empty by construction

6.4 Function definition: Extend a quad (used in phase 2)

Starting from $Q_i = \begin{bmatrix} n_{1,i} & n_{1,i+1} \\ n_{2,i} & n_{2,i+1} \end{bmatrix}$, which must be a valid quad in a structured region, we would like to find two more vertices $n_{1,i+2}$ and $n_{2,i+2}$, such that $Q_{i+1} = \begin{bmatrix} n_{1,i+1} & n_{1,i+2} \\ n_{2,i+1} & n_{2,i+2} \end{bmatrix}$ forms a valid quad in a structured quad region. They must satisfy the following constraints:

- Each of the vertices $n_{1,i+2}$ and $n_{2,i+2}$ must have exactly 4 neighbours.
- Each of the following pairs of vertices are neighbours: $n_{1,i+1}$ and $n_{1,i+2}$; $n_{2,i+1}$ and $n_{2,i+2}$; $n_{1,i+2}$ and $n_{2,i+2}$.
- Each of the vertices $n_{1,i+2}$ and $n_{2,i+2}$ must *not* neighbour any vertex that has been visited thus far, apart from those vertices explicitly mentioned.



6.4.1 Algorithm

1. $n_{1,i+1}$ has 4 neighbours, which include $n_{1,i}$ and $n_{2,i+1}$. Call the remaining 2 neighbours a and b. This is expressed by:

$$Adj_{VV}(n_{1,i+1}) \setminus \{n_{1,i}, n_{2,i+1}\} = \{a, b\}$$

2. $n_{2,i+1}$ has 4 neighbours, which include $n_{2,i}$ and $n_{1,i+1}$. Call the remaining 2 neighbours c and d.

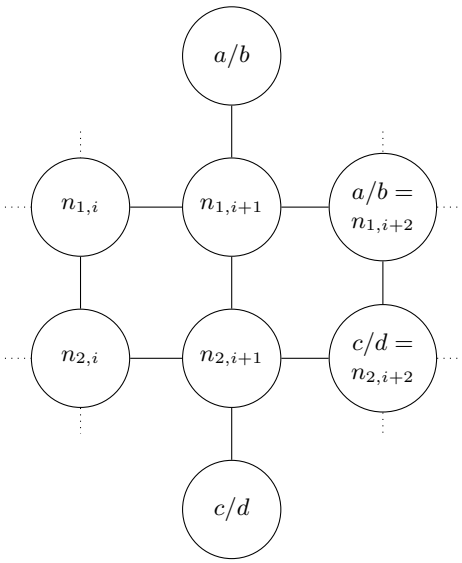
$$Adj_{VV}(n_{2,i+1}) \setminus \{n_{2,i}, n_{1,i+1}\} = \{c, d\}$$

3. Choose two vertices $n_{1,i+2} \in \{a, b\}$ and $n_{2,i+2} \in \{c, d\}$ such that $n_{1,i+2}$ and $n_{2,i+2}$ are neighbours, which is expressible[2] as:

$$n_{1,i+2} \in Adj_{VV}(n_{2,i+2})$$

[2]Or equivalently (by symmetry of Adj_{VV}) $n_{2,i+2} \in Adj_{VV}(n_{1,i+2})$

and such that they each have exactly 4 neighbours. If no such vertices $n_{1,i+2}$ and $n_{2,i+2}$ exist, fail the procedure.



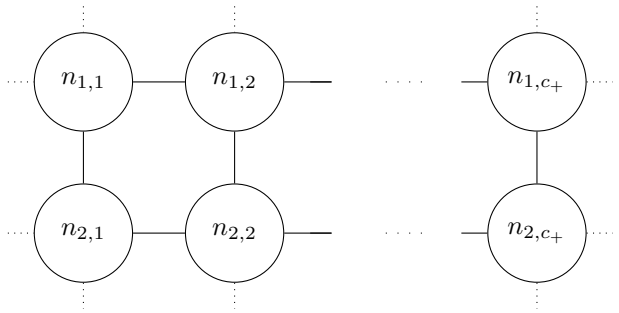
4. If any of the two vertices $n_{1,i+2}$ and $n_{2,i+2}$ exists in $V_{visited}$, that is $\{n_{1,i+2}, n_{2,i+2}\} \cap V_{visited} \neq \varnothing$, fail the procedure. Otherwise add the two vertices to $V_{visited}$.

5. Ensure for every vertex $n \in \{n_{1,i+2}, n_{2,i+2}\}$ that its visited neighbours, $Adj_{VV}(n) \cap V_{visited}$, are exactly those explicitly stated above. If this is not the case, remove $\{n_{1,i+2}, n_{2,i+2}\}$ from $V_{visited}$ and fail the procedure.

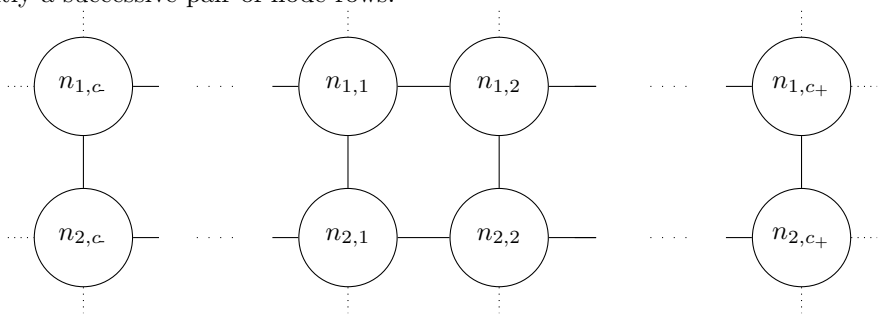
6. Return the two vertices $n_{1,i+2}$ and $n_{2,i+2}$.

6.5 Phase 2: Extend a row

"Extend a quad" is iteratively applied, starting from $Q_1 = \begin{bmatrix} n_{1,1} & n_{1,2} \\ n_{2,1} & n_{2,2} \end{bmatrix}$, to yield successive quads until it fails. The resulting vertices are appended to the *right* of $V_{structured}$ to form a single quad row in a structured quad region, or equivalently a successive pair of node rows.



Next, "Extend a quad" is again iteratively applied, this time with $Q_1 = \begin{bmatrix} n_{1,2} & n_{1,1} \\ n_{2,2} & n_{2,1} \end{bmatrix}$, the mirror image of Q_{init} about the y-axis. The repeated application yields successive quads until the procedure fails. The resulting vertices are appended to the *left* of $V_{structured}$ to extend the existing quad row, or equivalently a successive pair of node rows.



6.5.1 Algorithm

Extend to the right

1. Let $Q_{in} = Q_{init} = \begin{bmatrix} n_{1,1} & n_{1,2} \\ n_{2,1} & n_{2,2} \end{bmatrix}$, as produced by "Grow a quad".
2. Let $\begin{bmatrix} n_{1,i} & n_{1,i+1} \\ n_{2,i} & n_{2,i+1} \end{bmatrix}$ represent the elements of Q_{in}.
3. Call the procedure "Extend a quad" with Q_{in} as input.
4. If the procedure fails, go to step 7.
5. Otherwise, append the two new vertices obtained, $n_{1,i+2}$ and $n_{2,i+2}$ to the *right* of $V_{structured}$.
   Thus $V_{structured}$ will become: $\begin{bmatrix} n_{1,1} & n_{1,2} & \cdots & n_{1,i} & n_{1,i+1} & n_{1,i+2} \\ n_{2,1} & n_{2,2} & \cdots & n_{2,i} & n_{2,i+1} & n_{2,i+2} \end{bmatrix}$
6. Set $Q_{in} = \begin{bmatrix} n_{1,i+1} & n_{1,i+2} \\ n_{2,i+1} & n_{2,i+2} \end{bmatrix}$, and continue from step 2.

Extend to the left

7. Let $Q_{in} = \begin{bmatrix} n_{1,2} & n_{1,1} \\ n_{2,2} & n_{2,1} \end{bmatrix}$ be the mirror image of Q_{init} about the y-axis.

8. Let $\begin{bmatrix} n_{1,i} & n_{1,i+1} \\ n_{2,i} & n_{2,i+1} \end{bmatrix}$ represent the elements of Q_{in}.

9. Call the procedure "Extend a quad" with Q_{in} as input.

10. If the procedure fails, return immediately.

11. Otherwise, append the two new vertices obtained, $n_{1,i+2}$ and $n_{2,i+2}$ to the *left* of $V_{structured}$.
    Thus $V_{structured}$ will become: $\begin{bmatrix} n_{1,i+2} & n_{1,i+1} & n_{1,i} & \cdots & n_{1,1} & n_{1,2} & \cdots & n_{1,c} \\ n_{2,i+2} & n_{2,i+1} & n_{2,i} & \cdots & n_{2,1} & n_{2,2} & \cdots & n_{2,c} \end{bmatrix}$
    where $n_{1,c}$ and $n_{2,c}$ denote the rightmost vertices in $V_{structured}$.

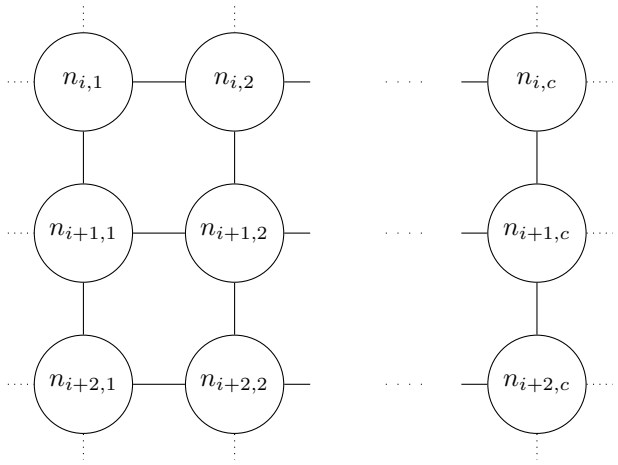
12. Set $Q_{in} = \begin{bmatrix} n_{1,i+1} & n_{1,i+2} \\ n_{2,i+1} & n_{2,i+2} \end{bmatrix}$, and continue from step 8.

6.6 Function definition: Extend rows (used in phase 3)

We are given a quad row in a structured quad region, that is a pair of successive node rows, call them r_i and r_{i+1}. The nodes in each row are denoted by $n_{i,1} \cdots n_{i,c}$ and $n_{i+1,1} \cdots n_{i+1,c}$, respectively, where $c >= 3$ is the number of node columns.

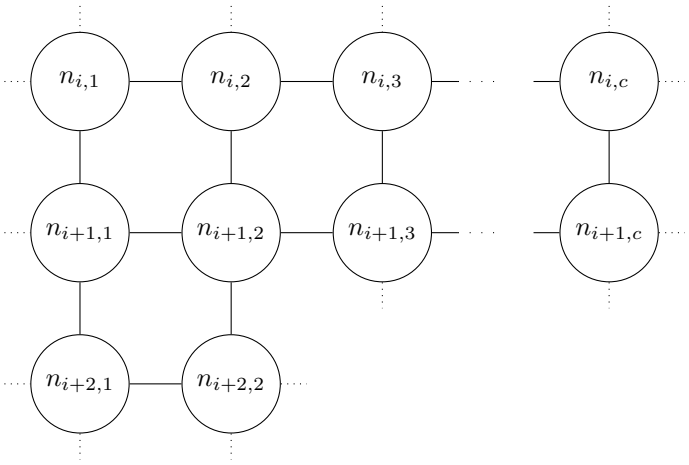
We extend this pair of node rows r_i and r_{i+1} by a third node row, r_{i+2}, consisting of nodes $n_{i+2,1} \cdots n_{i+2,c}$. These must satisfy the following conditions:

- Each vertex $n_{i+2,j}$ for $j \in [1..c]$ must have exactly four neighbours.
- Each of the pairs of vertices $n_{i+2,j}$ and $n_{i+2,j+1}$ for $j \in [1..c-1]$ are neighbours.
- Each of the pairs of vertices $n_{i+1,j}$ and $n_{i+2,j}$ for $j \in [1..c]$ are neighbours.
- Each vertex $n_{i+2,j}$ for $j \in [1..c]$ must not neighbour any vertex that has been visited thus far, apart from those vertices explicitly mentioned.



6.6.1 Algorithm

1. Let $n_{i+2,2}$ be the unique vertex in the set $Adj_{VV}(n_{i+1,2}) \backslash \{n_{i+1,1}, n_{i+1,3}, n_{i,2}\}$.
2. Let $N = Adj_{VV}(n_{i+1,1}) \cap Adj_{VV}(n_{i+2,2})$ be the common neighbours of $n_{i+1,1}$ and $n_{i+2,2}$. If the number of common neighbours $|N|$ is not exactly two, fail the procedure.
3. We know that $n_{i+1,2} \in N$ by construction. Let $n_{i+2,1}$ be the other vertex in N.
4. If either of $n_{i+2,1}$ or $n_{i+2,2}$ is in $V_{visited}$, fail the procedure. Otherwise add the two vertices to $V_{visited}$.
5. Ensure that the visited neighbours of $n_{i+2,1}$ and $n_{i+2,2}$ are exactly as specified, that is $n_{i+2,1} \cap V_{visited} = \{n_{i+2,2}, n_{i+1,1}\}$ and $n_{i+2,2} \cap V_{visited} = \{n_{i+2,1}, n_{i+1,2}\}$. If this is not the case, remove $n_{i+2,1}$ and $n_{i+2,2}$ from $V_{visited}$ and fail the procedure.

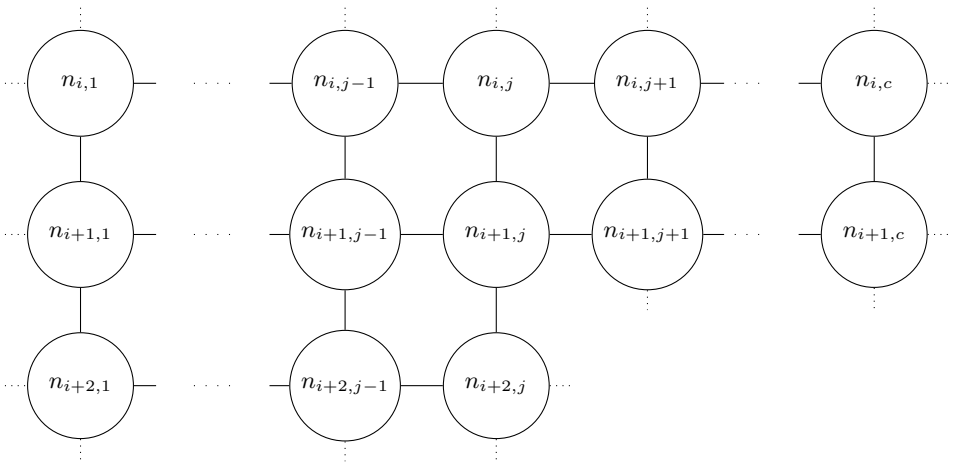


6. For each column[3] $j \in [3..c-1]$:

   (a) Let $n_{i+2,j}$ be the unique vertex in the set $Adj_{VV}(n_{i+1,j}) \backslash \{n_{i+1,j-1}, n_{i+1,j+1}, n_{i,j}\}$.

   (b) If $n_{i+2,j} \in V_{visited}$, remove all vertices in r_{i+2} from $V_{visited}$, that is $\{n_{i+2,k} \mid k \in [1..j]\}$ and fail the procedure. Otherwise, add $n_{i+2,j}$ $V_{visited}$.

   (c) Ensure that visited neighbours of $n_{i+2,j}$ are exactly $n_{i+2,j-1}$, $n_{i+1,j}$. If this is not the case remove vertices $\{n_{i+2,k} \mid k \in [1..j]\}$ from $V_{visited}$ and fail the procedure.



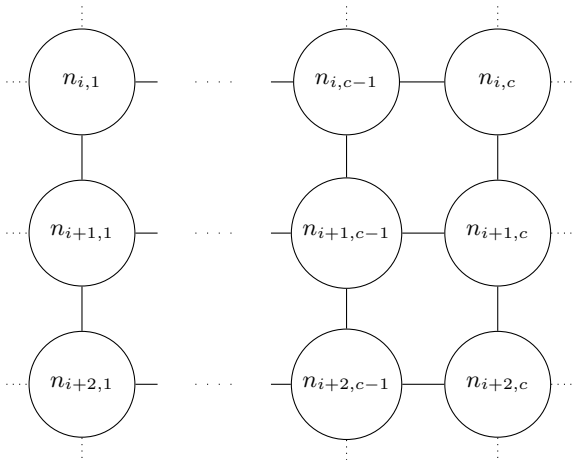
7. Consider $Adj_{VV}(n_{i+2,c-1}) \cap Adj_{VV}(n_{i+1,c})$, the common neighbours of $n_{i+2,c-1}$ and $n_{i+1,c}$. If these are not exactly two neighbours, remove vertices $\{n_{i+2,k} \mid k \in [1..c-1]\}$ from $V_{visited}$ and fail the procedure.

8. One of the two neighbours must be $n_{i+1,c}$ by construction. Let $n_{i+2,c}$ denote the other neighbour.

9. If $n_{i+2,c} \in V_{visited}$, remove vertices $\{n_{i+2,k} \mid k \in [1..c-1]\}$ from $V_{visited}$ and fail the procedure. Otherwise, add $n_{i+2,c}$ to $V_{visited}$.

10. Ensure that the visited neighbours of $n_{i+2,c}$ are exactly $\{n_{i+2,c-1}, n_{i+1,c}\}$, that is $Adj_{VV}(n_{i+2,c}) \cap V_{visited} = \{n_{i+2,c-1}, n_{i+1,c}\}$. If this is not the case remove vertices $\{n_{i+2,k} \mid k \in [1..c]\}$ from $V_{visited}$ and fail the procedure.

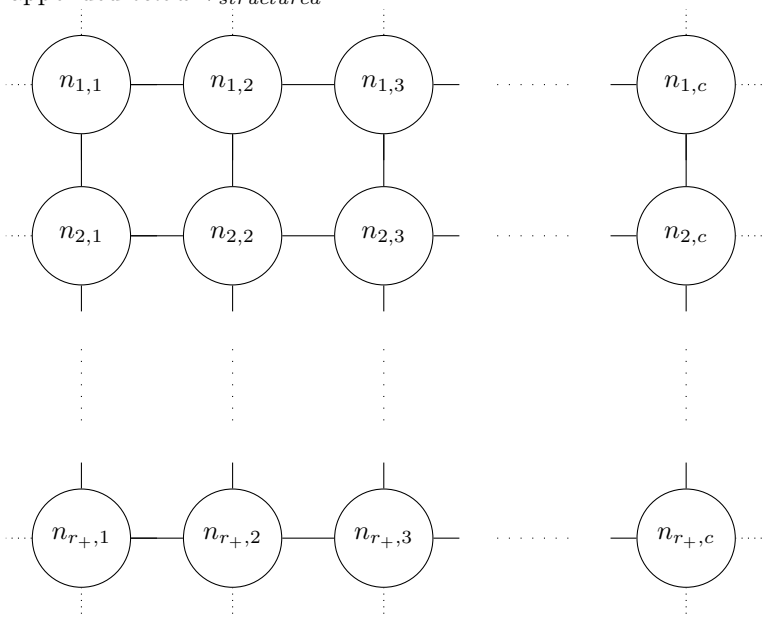
[3] Possibly none if $c = 3$



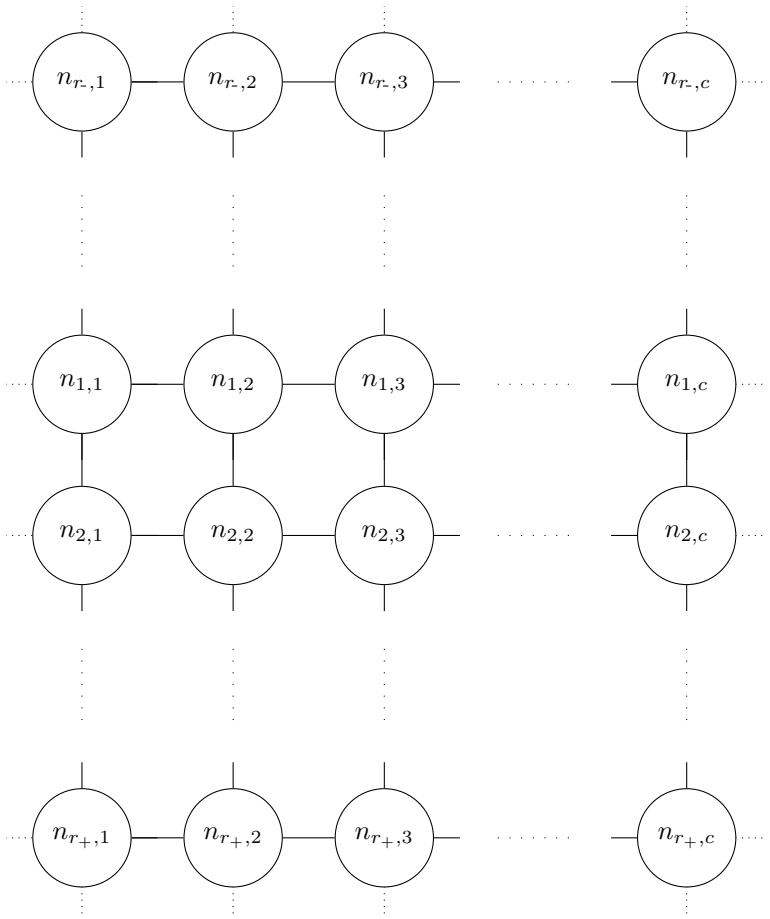
11. Return the vertices of node row r_{i+2}, that is $n_{i+2,1} \cdots n_{i+2,c}$.

6.7 Phase 3: Extend rows

"Extend a row" is iteratively applied, starting from $r_1 = n_{1,1} \cdots n_{1,c}$ and $r_2 = n_{2,1} \cdots n_{2,c}$, to yield successive rows until it fails. The resulting rows are appended *below* $V_{structured}$.



"Extend a row" is iteratively applied, starting from the mirror image of the initial rows about the x-axis, that is $r_1 = n_{2,1} \cdots n_{2,c}$ and $r_2 = n_{1,1} \cdots n_{1,c}$, to yield successive rows until it fails. The resulting rows are appended *above* $V_{structured}$.



6.7.1 Algorithm

Extend below

1. Let $r_a = n_{1,1} \cdots n_{1,c}$ and $r_b = n_{2,1} \cdots n_{2,c}$ be the first two structured node rows.

2. Let $n_{i,1} \cdots n_{i,c}$ and $n_{i+1,1} \cdots n_{i+1,c}$ represent the respective elements of r_a and r_b.

3. Call the procedure "Extend a row" with r_a and r_b as inputs.

4. If the procedure fails, go to step 7.

5. Otherwise, append the new row obtained $r_{new} = n_{i+2,1} \cdots n_{i+2,c}$ *below* $V_{structured}$.

Thus $V_{structured}$ will become: $\begin{bmatrix} n_{1,1} & n_{1,2} & \cdots & n_{1,c} \\ n_{2,1} & n_{2,2} & \cdots & n_{2,c} \\ \vdots & \vdots & & \vdots \\ n_{i,1} & n_{i,2} & \cdots & n_{i,c} \\ n_{i+1,1} & n_{i+1,2} & \cdots & n_{i+1,c} \\ n_{i+2,1} & n_{i+2,2} & \cdots & n_{i+2,c} \end{bmatrix}$

6. Set $r_a = n_{i+1,1} \cdots n_{i+1,c}$ and $r_b = n_{i+2,1} \cdots n_{i+2,c}$, and continue from step 2.

Extend above

7. Let $r_a = n_{2,1} \cdots n_{2,c}$ and $r_b = n_{1,1} \cdots n_{1,c}$ be the first two structured node rows, the other way around.

8. Let $n_{i,1} \cdots n_{i,c}$ and $n_{i+1,1} \cdots n_{i+1,c}$ represent the respective elements of r_a and r_b.

9. Call the procedure "Extend a row" with r_a and r_b as inputs.

10. If the procedure fails, return immediately.

11. Otherwise, append the new row obtained $r_{new} = n_{i+2,1} \cdots n_{i+2,c}$ *above* $V_{structured}$.

    Thus $V_{structured}$ will become: $\begin{bmatrix} n_{i+2,1} & n_{i+2,2} & \cdots & n_{i+2,c} \\ n_{i+1,1} & n_{i+1,2} & \cdots & n_{i+1,c} \\ n_{i,1} & n_{i,2} & \cdots & n_{i,c} \\ \vdots & \vdots & & \vdots \\ n_{1,1} & n_{1,2} & \cdots & n_{1,c} \\ n_{2,1} & n_{2,2} & \cdots & n_{2,c} \\ \vdots & \vdots & & \vdots \\ n_{r_+,1} & n_{r_+,2} & \cdots & n_{r_+,c} \end{bmatrix}$

12. Set $r_a = n_{i+1,1} \cdots n_{i+1,c}$ and $r_b = n_{i+2,1} \cdots n_{i+2,c}$, and continue from step 8.

6.8 Sketch proof of complexity analysis

We argue that the length-first search algorithm presented in this chapter runs in $O(V_{structured})$, that is linear in the size of the detected structured region[4].

Neighbour queries are performed using the relation-maps available, and are hence assumed to run in $O(1)$. Set operations such as set-intersection and set-difference between a constant neighbours also run in $O(1)$, as we assume some (again small) constant upper bound on the number of neighbours.

[4]If no structured region is detected, then we take this to mean that the run-time should be constant, $O(1)$.

Querying the visited set or adding a vertex to it is done in $O(1)$.

If the algorithm terminates having not found any structure region, which can only occur in the *grow a quad* phase, then it would have only examined a constant number of neighbours and applied a constant number of set-operations.

If the algorithm terminates having found a structured region, then it must have called the "extend a quad" and "extend rows" functions a number of times linear in $V_{structured}$. We know this since each call to those functions adds a new vertex to $V_{structured}$. All other remaining work performed is linear in time.

Thus the time complexity of the length-first search algorithm is $O(V_{structured})$.

6.8.1 A corollary

Given that length-first search runs in $O(V_{structured})$, we show that using it for detecting multiple structure regions results in an $O(V)$ time complexity.

For every application of length-first search, we detect $V_{structured}$ vertices in $O(V_{structured})$ time. Vertices which are detected are added to the visited set and hence are not used as a starting vertex in the future. Since length-first search consumes as many vertices as it consumes time (asymptotically speaking), then by the end we would have exhausted all vertices, and have done so in $O(V)$.

6.9 Chapter summary

In this chapter we refined our sketch of the length-first search algorithm into a more precise form more amenable for complexity analysis. On this basis we analyse the algorithmic complexity of the algorithm, and find it to be linear in the number of vertices.

Chapter 7

Exposing structure

Having identified the structured regions in the mesh, we discuss how this structure may be *exposed* at the data layout level.

7.1 Defining structured region boundaries

When running a core-computation over structured data, we would like to ensure that direct neighbour accesses are always within bounds. To this end, we define the concepts of *interior* structured elements and *fringe* structured elements, with respect to a given relation-map between a set and itself. In subsection 7.3.1 we generalise the definition to encompass relation-maps between arbitrary sets.

Let $R : S \mapsto S$ be a relation-map on an element set S. We assume that S has been partitioned into k structured regions S_1 to S_k, as well as the remaining unstructured partition S_0. Let $e \in S_i$ be some structured mesh element in some structured region S_i.

We say that e is an *interior structured element* iff all neighbours n of e, as defined by R, are structured mesh elements in S_i. Stated formally:

$$\forall n \in S.R(e, n) \implies n \in S_i$$

Conversely, a *fringe structured element* is one which does not satisfy this property, that is it has some neighbour in R which is not found within the structured region S_i.

Figure 7.1 exemplifies this for structured cell regions.

7.2 Laying out vertices

With the regions of structured vertices extracted, we would like to lay out the associated vertex data in a convenient order such that the neighbours' data may be accessed directly. Recalling our decision in 4.1.2, we choose to detect rectangular structured regions so as to place them in two-dimensional arrays.

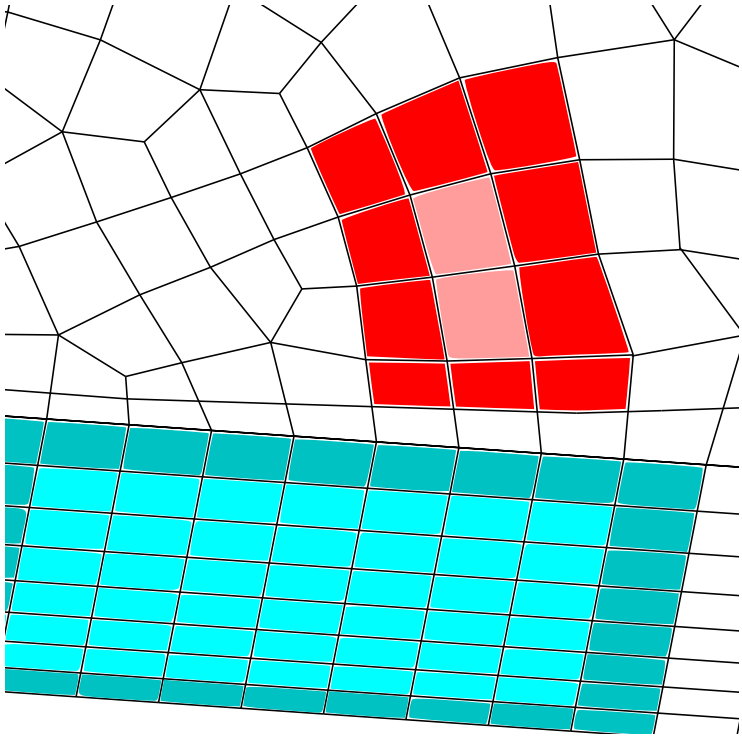
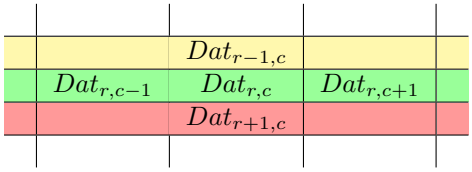


Figure 7.1: The image depicts the boundaries induced by a cell-cell relation-map. The two structured regions are coloured in blue and red. Lighter shades denote interior structured cells, and darker shades denote fringe structured cells.



(a) Logical layout of the *Dat* array. Neighbours are horizontally and vertically contiguous.

...	$Dat_{r-1,c}$	...	$Dat_{r,c-1}$	$Dat_{r,c}$	$Dat_{r,c+1}$	...	$Dat_{r+1,c}$	...

(b) Physical layout of the *Dat* array. The contiguous data elements are contiguous in memory.

Figure 7.2: The layout the *Dat* array, representing vertex-vertex associated data. This data stored may represent the spatial coordinates of vertices, for example.

This still leaves us to determine the data layout within a two-dimensional array. We use a row-major order as it enables very simple (cheap) neighbour address calculations; column-major order exhibits the same property and may been used equally. Alternative orderings, as well as the general description of space filling curves are described in detail in [25], and may be worth exploring in future works.

Then given a rectangular structured vertex region $V_{structured}$ with m rows and n columns, we can represent its associated data in a two-dimensional array *Dat* in row-major order. Clearly, the associated data of *any* structured vertex $n_{r,c} \in V_{structured}$ can be accessed by $Dat[r][c]$. Similarly, we can access the associated data of the vertex neighbours of any *interior* structured vertex $n_{r,c} \in V_{structured}$ by $Dat[r][c+1]$, $Dat[r][c-1]$, $Dat[r+1][c]$, and $Dat[r-1][c]$. Figure 7.2 illustrates the data layout and the resultant memory access pattern.

7.3 Overlaying the remaining mesh elements

Our construction thus far only allows us to directly address data via a vertex-vertex relation, and we would like to extend the benefits to include other relation-maps. The key insight is the knowledge that relation-maps do not define an arbitrary relation, rather they *represent compositional relationships in the mesh hierarchy*:

- A structured vertex $n_{r,c}$ forms a horizontal edge with $n_{r,c+1}$, and a vertical edge with $n_{r+1,c}$.
- Four structured vertices of the given relative positions form the vertices of a quadrilateral cell: $n_{r,c}$, $n_{r,c+1}$, $n_{r+1,c}$, and $n_{r+1,c+1}$.
- Four structured vertex pairs of the given relative positions form the edges of a quadrilateral cell: $n_{r,c}$ with $n_{r,c+1}$, $n_{r,c+1}$ with $n_{r+1,c+1}$, $n_{r+1,c+1}$

with $n_{r+1,c}$, and $n_{r+1,c}$ with $n_{r,c}$.

We can then derive for each structured vertex region a) a structured cell region, b) a structured horizontal-edges region, and c) a structured vertical-edge region. We refer to a given structured vertex region and the structured regions derived from it as *corresponding structured regions*.

The edge case is an interesting one: the reason that we derive two structured regions, horizontal and vertical, is that they exhibit a different neighbour access pattern.

In addition, as with vertex associated data, the associated data of each of the derived structured regions may be laid out in any convenient order; we continue with the same row-major order.

7.3.1 Redefining structured region boundaries: the general case

Given the above information, we can generalize our definition of interior and fringe structured elements to allow for relation-maps between different element sets.

Let $R : S \mapsto T$ be a relation-map between element sets S and T. We assume that S and T have each been partitioned into k mutually *corresponding* structured regions S_1 to S_k and T_1 to T_k, respectively, where each structured region S_i corresponds to structured region T_i. Additionally, S_0 and T_0 represent the unstructured partitions in each of S and T[1]. Let $e \in S_i$ be some structured mesh element in some structured region S_i.

We say that e is an *interior structured element* iff all neighbours n of e, as defined by R, are structured mesh elements in the corresponding structured region T_i. Stated formally:

$$\forall n \in T.R(e, n) \implies n \in T_i$$

Conversely, a *fringe structured element* is one which does not satisfy this property, that is it has some neighbour in R which is not found within the corresponding structured region T_i.

Figure 7.3 shows an example for an edge-to-cell relation-map.

7.3.2 Identifying element numbering

Deriving an overlaid structured region is simple, and allows for direct access to neighbouring elements. This derivation method, however, does not reveal the implicitly defined partitioning of mesh elements (into structured and unstructured mesh elements), in particular we do not know how to iterate over the unstructured mesh elements, as demonstrated in figure 7.4 for the case of a cell-cell relation-map.

[1] As per our definition, S_0 and T_0 cannot be corresponding structured regions, as they are not structured!

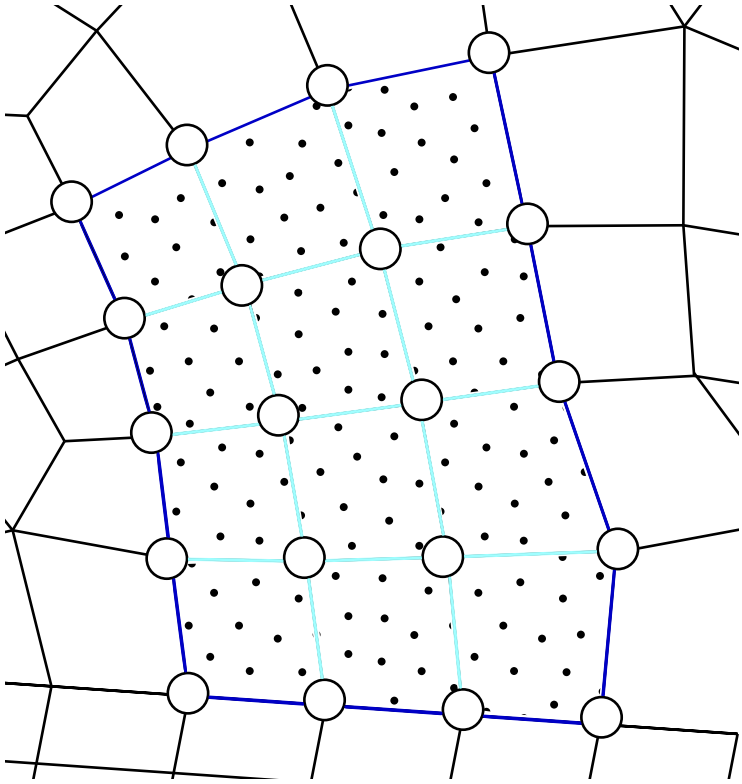


Figure 7.3: The image depicts the boundaries induced by an edge-to-cell relation-map. Fringe structured edges (having only one structured cell neighbour) are highlighted in dark blue, and interior structured edges (having both neighbours structured cells) are highlighted in light blue. All structured cells are are dotted. Structured vertices, represented by circles, demarcate the structured vertex region.

If we can find the element ids corresponding to the derived structured elements, we can then deduce contents of the other partition: the unstructured elements. For this we require a relation-map from vertices to the element type in question, so then we can map the vertices forming the derived mesh element to its respective element id. This relation-map may already be provided, or we may need to derive it from other relation-maps.

As an example, consider the relation-map in figure 7.5 mapping each cell to a unique set of four vertices. If we invert this map, then we obtain a partial map from sets of four vertices to a unique cell, each. Since the map is indexed by sets of vertices, a direct-indexed array is not an appropriate choice of data structure, and the use of, for example, a hash table would be required. Alternatively, we may represent the relation-map as a map Adj_{VC} mapping each vertex to a set of cells adjacent to it. Determining the unique cell (if any) composed of four vertices $\{n_1, n_2, n_3, n_4\}$ can then be determined by the following:

$$Adj_{VC}(n_1) \cap Adj_{VC}(n_2) \cap Adj_{VC}(n_3) \cap Adj_{VC}(n_4)$$

Figure 7.6 shows examples of the two relation-map inversion methods.

With the required relation-map (vertices to element type) available, we can straightforwardly determine the element ids of the fringe structured elements and the unstructured elements, and hence the variable remaining_cell2cells from figure 7.4.

We hasten to point out despite the linear complexity of creating and accessing an inverted map, in practice the costs associated with set operations and disparate memory accesses tend to be high for large meshes unless very carefully optimised. In fact, much of the runtime costs incurred by our structure detection implementation, described in chapter 9, are due to the structure overlaying process.

```
for (cell_id, neighbour_cells_ids) in cell2cells:
  kernel_function(cell_id, neighbour_cells_ids)
```

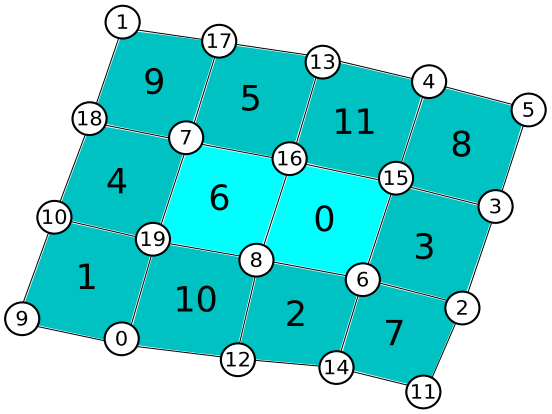
(a) Applying a kernel function over the original cell-cell relation-map. Variable cell2cells is the cell-cell relation-map.

```
for region in structured_regions:
  for row in region.interior_structured_rows:
    for column in region.interior_structured_cols:
      # Compute cell ids ... (details omitted)
      kernel_function(cell_id, neighbour_cells_ids)

remaining_cell2cells = ... # this is currently unknown
for (cell_id, neighbour_cells_ids) in remaining_cell2cells:
  kernel_function(cell_id, neighbour_cells_ids)
```

(b) Applying a kernel function over the interior structured cells, followed by the remaining cells. Notice how no relation-map is required for the former loop. Variable remaining_cell2cells is a presently unknown subset of the cell-cell relation-map which excludes interior structured cells.

Figure 7.4: A pseudo-code comparison between the original iteration over cells (7.4a), and the split iterations over the interior structured cells and the remaining cells (7.4b). Not knowing the iteration space of the remaining cells loop demonstrates the need to determine the structure-induced partitioning.



cell $\mapsto$ vertices
$c_0 \mapsto \{n_{16}, n_{15}, n_8, n_6\}$
$c_1 \mapsto \{n_{10}, n_{19}, n_9, n_0\}$
$c_2 \mapsto \{n_8, n_6, n_{12}, n_{14}\}$
$c_3 \mapsto \{n_{15}, n_3, n_6, n_2\}$
$c_4 \mapsto \{n_{18}, n_7, n_{10}, n_{19}\}$
$c_5 \mapsto \{n_{13}, n_{13}, n_7, n_{16}\}$
$c_6 \mapsto \{n_7, n_{16}, n_{19}, n_8\}$
$c_7 \mapsto \{n_6, n_2, n_{14}, n_{11}\}$
$c_8 \mapsto \{n_4, n_5, n_{15}, n_3\}$
$c_9 \mapsto \{n_1, n_{17}, n_{18}, n_7\}$
$c_{10} \mapsto \{n_{19}, n_8, n_0, n_{12}\}$
$c_{11} \mapsto \{n_{13}, n_4, n_{16}, n_{15}\}$

Figure 7.5: A small example mesh consisting of cells and vertices, along with a cell-vertex relation-map. As before, lighter shades denote interior structured cells, and darker shades denote fringe structured cells.

vertex $\mapsto$ cells
$n_0 \mapsto \{c_1, c_{10}\}$
$n_1 \mapsto \{c_9\}$
$n_2 \mapsto \{c_3, c_7\}$
$n_3 \mapsto \{c_8, c_3\}$
$n_4 \mapsto \{c_8, c_{11}\}$
$n_5 \mapsto \{c_8\}$
$n_6 \mapsto \{c_0, c_2, c_3, c_7\}$
$n_7 \mapsto \{c_9, c_4, c_5, c_6\}$
$n_8 \mapsto \{c_0, c_2, c_{10}, c_6\}$
$n_9 \mapsto \{c_1\}$
$n_{10} \mapsto \{c_1, c_4\}$
$n_{11} \mapsto \{c_7\}$
$n_{12} \mapsto \{c_2, c_{10}\}$
$n_{13} \mapsto \{c_{11}, c_5\}$
$n_{14} \mapsto \{c_2, c_7\}$
$n_{15} \mapsto \{c_0, c_8, c_3, c_{11}\}$
$n_{16} \mapsto \{c_0, c_{11}, c_5, c_6\}$
$n_{17} \mapsto \{c_9\}$
$n_{18} \mapsto \{c_9, c_4\}$
$n_{19} \mapsto \{c_1, c_{10}, c_4, c_6\}$

(a) A map from vertices to sets of cells. The unique cell, if any, corresponding to a set of four vertices can be determined by computing the intersection of cell-sets obtained.

vertex-set $\mapsto$ cell
$\{n_{16}, n_{15}, n_8, n_6\} \mapsto c_0$
$\{n_{10}, n_{19}, n_9, n_0\} \mapsto c_1$
$\{n_8, n_6, n_{12}, n_{14}\} \mapsto c_2$
$\{n_{15}, n_3, n_6, n_2\} \mapsto c_3$
$\{n_{18}, n_7, n_{10}, n_{19}\} \mapsto c_4$
$\{n_{13}, n_{13}, n_7, n_{16}\} \mapsto c_5$
$\{n_7, n_{16}, n_{19}, n_8\} \mapsto c_6$
$\{n_6, n_2, n_{14}, n_{11}\} \mapsto c_7$
$\{n_4, n_5, n_{15}, n_3\} \mapsto c_8$
$\{n_1, n_{17}, n_{18}, n_7\} \mapsto c_9$
$\{n_{19}, n_8, n_0, n_{12}\} \mapsto c_{10}$
$\{n_{13}, n_4, n_{16}, n_{15}\} \mapsto c_{11}$

(b) A partial map from sets of four vertices to unique cells. A data structure such as a hash table is needed to index using sets.

Figure 7.6: Two methods of building an inverse mapping for the cell-vertex relation-map in figure 7.5.

7.4 Renumbering elements

So far we have only considered neighbour accesses for interior structured elements; this leaves us with two classes of mesh elements:

- Fringe structured mesh elements.

  These may have both structured and unstructured neighbours (or potentially non-neighbours if the element is at a *mesh* boundary). They are accessed directly via address calculation by their interior structured neighbours, and indirectly via relation-maps by their fringe structured neighbours[2] and their unstructured neighbours (if any).

[2]In principle, fringe structured elements can access their fringe neighbours via address calculation so long as they fall within the same structured region. This however, would require a "mixed neighbour-addressing" mode for fringe elements, which complicates the addressing logic.

- Unstructured mesh elements.

  These always access (and are accessed by) their neighbours indirectly via relation-maps.

Observe that mesh elements that are accessed directly, the structured elements in other words, have an imposed storage location. This is the reason why structured regions must be disjoint, otherwise multiple structured region would impose conflicting storage locations on the same element[3]. Mesh elements that are indirectly accessed on the other hand are not constrained in their placement. Thus we can treat fringe structured mesh elements as structured in terms of their storage placement, and as unstructured when accessing their neighbours.

We can enumerate all the possible neighbour access types as follows (see figure 7.7 for illustrations). We later discuss which neighbour access types would be most beneficial to use.

a) Two interior structured elements. Both elements must be co-located in the same structured region, and are directly addressable. No further data needs to be stored.

b) An interior structured element and a fringe structured element. Same as above.

c) Two fringe structured elements. The two elements may or may not fall in the same structured region. In the former case, the above applies. In the latter case, which may occur if two structured regions are adjacent, the elements must be indirectly addressed.

d) A fringe structured element and an unstructured element. The two elements are not co-located and must be indirectly addressed.

e) Two unstructured elements. The two elements are co-located in the same partition, but must be indirectly addressed.

We would like to minimize the cost and complication of cross-region neighbour accesses, which arise due to fringe structured elements. To this avail, all regions for the given associative data, structured and unstructured, are stored in a single address space. All structured regions are stored sequentially in the order of their discovery[4], followed by the remaining unstructured elements. Figures 7.8 and 7.9 show an example of renumbering applied to cells.

The relation-map can then simply store indices to denote each element's neighbours. In fact, if we maintain the explicit neighbour relations for *all* mesh elements, both structured and unstructured, a core-computation loop completely oblivious to our manipulations may execute over the relation-maps. The

[3]Whilst duplicating data would work for read-only access, at the expense of extra storage, this can be a disaster for writable data as duplicate data must be correctly synchronized.

[4]This is no compelling reason for this decision; this was merely done to simplify adding structured regions as they are discovered.

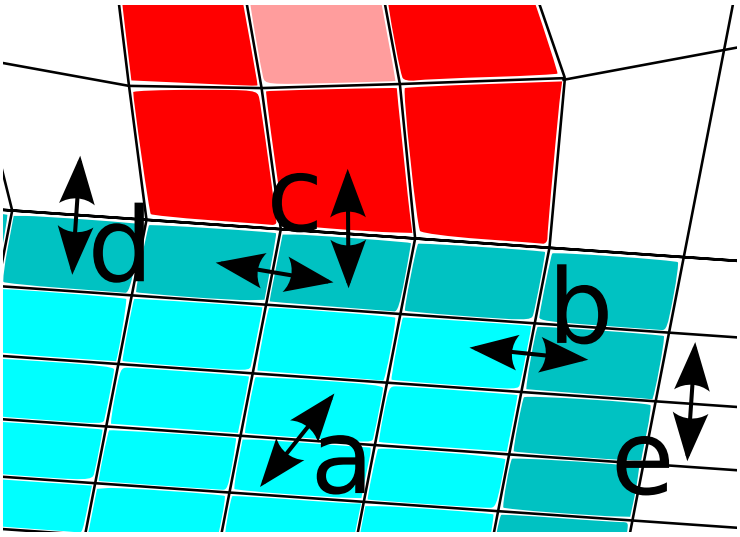
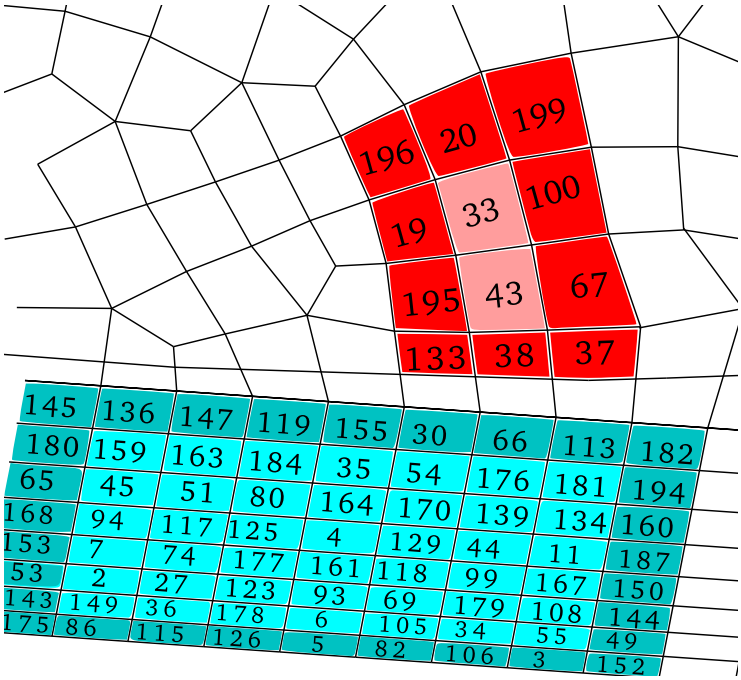
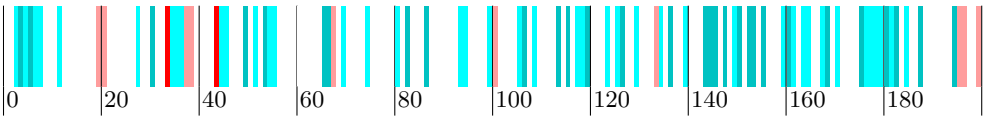


Figure 7.7: Examples of the different types neighbour access types. A double-arrow indicates that the two pointed-to cells have a neighbour relationship. The labels correspond to the list of possible neighbour accesses in section 7.4.

main downside to storing is the missed opportunity to completely exclude interior structured nodes from the relation map.

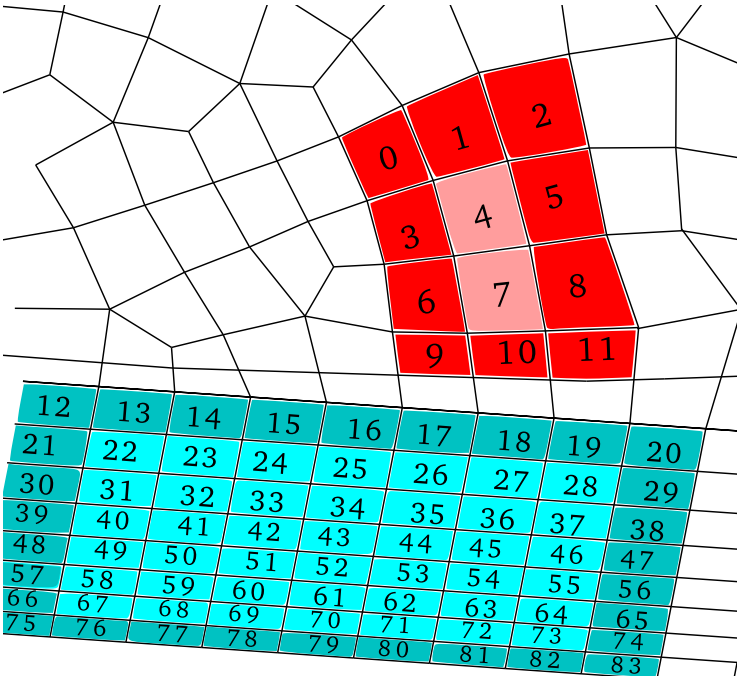


(a) Mesh showing the original cell numbering of the structured regions. Numbering of unstructured cells is omitted. Light and dark shades denote interior and fringe structured cells, respectively.

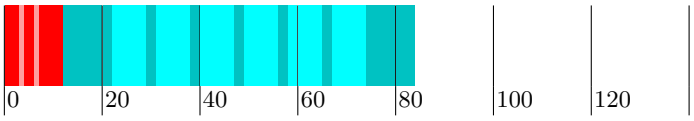


(b) The cell storage layout in memory due to the original cell numbering. The numbers (indicating cell ids), and the colours correspond to the mesh diagram above.

Figure 7.8: The original cell numbering. The boundaries shown are with respect to a *cell-cell* relation-map.



(a) A mesh with the Crystal cell numbering shown. Numbering of unstructured cells is omitted. Light and dark shades denote interior and fringe structured cells, respectively.



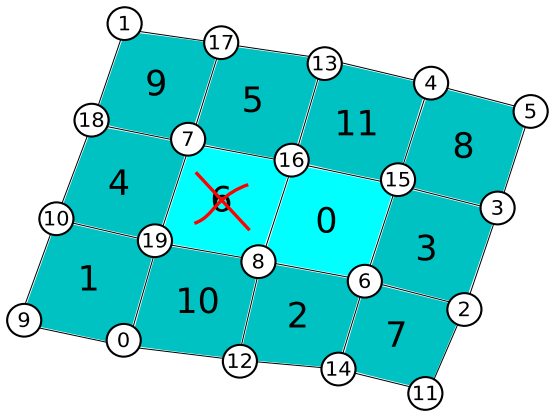
(b) The cell storage layout in memory due to the Crystal cell numbering. The numbers (indicating cell ids) and the colours correspond to the mesh diagram above.

Figure 7.9: Crystal cell numbering. The boundaries shown are with respect to a *cell-cell* relation-map.

7.5 Handling missing elements

The scheme described for deriving an overlaid structure works well when our assumptions about how the mesh hierarchy is composed holds uniformly. Whilst structured vertices are detected from first principles, the other structured mesh elements are overlaid on top, matched to their corresponding element ids. As an example, consider the case where four structured vertices logically *should* form a cell, in the manner we defined, but the cell-vertex relation-map indicates otherwise. This scenario is shown in figure 7.10.

We can handle this problem in a number of different ways, continuing the example involving missing cells:



cell $\mapsto$ vertices
$c_0 \mapsto \{n_{16}, n_{15}, n_8, n_6\}$
$c_1 \mapsto \{n_{10}, n_{19}, n_9, n_0\}$
$c_2 \mapsto \{n_8, n_6, n_{12}, n_{14}\}$
$c_3 \mapsto \{n_{15}, n_3, n_6, n_2\}$
$c_4 \mapsto \{n_{18}, n_7, n_{10}, n_{19}\}$
$c_5 \mapsto \{n_{13}, n_{13}, n_7, n_{16}\}$
~~$c_6 \mapsto \{n_7, n_{16}, n_{19}, n_8\}$~~
$c_7 \mapsto \{n_6, n_2, n_{14}, n_{11}\}$
$c_8 \mapsto \{n_4, n_5, n_{15}, n_3\}$
$c_9 \mapsto \{n_1, n_{17}, n_{18}, n_7\}$
$c_{10} \mapsto \{n_{19}, n_8, n_0, n_{12}\}$
$c_{11} \mapsto \{n_{13}, n_4, n_{16}, n_{15}\}$

Figure 7.10: The mesh from figure 7.5 with cell 6 missing, alongside its cell-vertex relation-map

- The mesh region is not structured to our satisfaction; the underlying structured vertex region (and any corresponding structured regions, such as a corresponding edge structured region) are dropped.
- The underlying vertex structured region is indeed structured, and can be used for accesses that do not involve cells. Only the overlaid structured cell region is dropped.
- The cells may be systematically missing near the fringe of the structured region. We can attempt to extract a sub-rectangle which does not contain any missing cells, and simply store the offsets defining that sub-rectangle.
- Arbitrarily positioned structured cells may be missing, but a large number may still be structured. We can skip over the missing gaps using a bit-mask, similar to the approach discussed in subsection 4.2.1.

We choose to handle missing elements by extracting a structured sub-rectangle. The approach involves minimum storage overhead, and is inexpensive to apply in both structured detection as well as in executing the core-computation .

7.6 Ensuring neighbour ordering

When iterating over a mesh entity set, such as the set of cells, we collect the needed indexing variables via a relation-map, for example vertices using a cell-vertex relation map. Recall the following specification of kernel functions in subsection 2.3.3:

> The indexing variables ... are passed to the kernel in some known order, typically in the order stored in the relation-map.

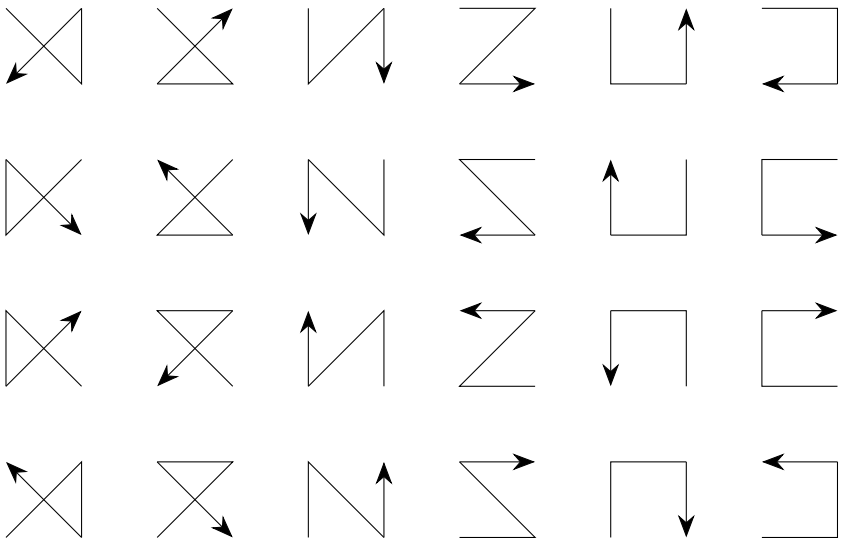


Figure 7.11: An enumeration of all possible cell-vertex neighbour ordering. Note the three families of orderings: $\sqsubset$, $\rtimes$, and $\sqsupset$.

In the case of unstructured neighbour accesses, this is a non-issue: the neighbours are stored in some explicit ordering in the relation-map (as part of the given mesh specification). In the case of structured neighboured accesses, however, the neighbours are derived in some order which is independent from the one defined by the relation-map. Figure 7.12 characterizes some examples of kernel functions where not respecting the order can be a problem.

We can represent all the possible different neighbour orderings as one of a fixed set of permutations. In the case of a cell-vertex relation-map, this would imply $4! = 24$ permutations, which can be stored in a mere 5 bits. The permutation of any particular element can be determined by comparing the structured region order with the order as stored in the corresponding relation-map. Figure 7.11 illustrates all the possible neighbour orderings for a cell-vertex relation-map.

If neighbour orderings are different *within* a structured region, we have to take one of the following approaches:

1. Store information about the ordering with each element.

2. Remove some elements from the structured region so as to maintain a uniform ordering within the structured region.This can be done in lieu of the discussion in section 7.5 regarding the handling missing elements.

Approach 1 does not seem very attractive, as it will incur both storage costs per element, and is likely to be expensive when executed in the core-computation. We foll approach 2 and, as with our handling of missing elements, we extract a uniformly-ordered sub-rectangle from the structured region.

If neighbour orderings are uniform within each structured region (either by our construction above, or by a happy coincidence), then we can store each structured region's ordering with its meta-data. When iterating over each structured region in the core-computation, the correct ordering can then be applied.

As a point of interest, notice how the different orderings in figure 7.11 fall into three families: $\sqsubset$, $\bowtie$, and $\angle$. Within each family, all the different orderings can be obtained through a series of rotations and reflections. If we apply such a series of transformations onto our structured region itself to match a canonical orientation, then we reduce our set of possible orderings to three, one for each family; in all likelihood, only one family of orderings will be used throughout a mesh.

We shall revisit this point later when we discuss the core-computation loop stuff in section 8.3.

7.7 Chapter summary

In this chapter we discussed tactics for effectively exposing the detected structure of the mesh. This covered methodologies for renumbering the mesh elements, and design decision for the structured regions' the meta-data. We also deliberated techniques for detecting overlaid structured regions and data structures to represent them, and discussed ways to manage the issues associated with overlaid structures.

```
def mean_kernel(cell_id, v0_id, v1_id, v2_id, v3_id):
  x0, x1, x2, x3 = xs[v0_id], xs[v1_id], xs[v2_id], xs[v3_id]
  y0, y1, y2, y3 = ys[v0_id], ys[v1_id], ys[v2_id], ys[v3_id]

  x_means[cell_id] = (x0 + x1 + x2 + x3) / 4
  y_means[cell_id] = (y0 + y1 + y2 + y3) / 4
```

(a) Pseudo-code of a kernel function which computes the mean point of each cell. It is completely independent of the vertex ordering, and we are free to present the vertices in an arbitrary order.

```
def area_kernel(cell_id, v0_id, v1_id, v2_id, v3_id):
  x0, x1, x2, x3 = xs[v0_id], xs[v1_id], xs[v2_id], xs[v3_id]
  y0, y1, y2, y3 = ys[v0_id], ys[v1_id], ys[v2_id], ys[v3_id]

  area =  (x0*y1 - y0*x1)
  area += (x1*y2 - y1*x2)
  area += (x2*y3 - y2*x3)
  area += (x3*y0 - y3*x0)
  areas[cell_id] = area / 2
```

(b) Pseudo-code of a kernel function which computes the area of each cell. It is only dependant on the *cyclic* ordering of vertices; that is to say, we may rotate the vertices whilst maintaining the relative order between them.

```
def slope_kernel(cell_id, v0_id, v1_id, v2_id, v3_id):
  x0, x1, x2, x3 = xs[v0_id], xs[v1_id], xs[v2_id], xs[v3_id]
  y0, y1, y2, y3 = ys[v0_id], ys[v1_id], ys[v2_id], ys[v3_id]

  slope0 = (y1 - y0) / (x1 - x0)
  slope1 = (y3 - y2) / (x3 - x2)

  slope_diffs[cell_id] = slope1 - slope0
```

(c) Pseudo-code of a kernel function which computes the relative difference in slope between two particular edges of each cell. It is completely dependant on the ordering of vertices, that is to say we may not alter the order.

Figure 7.12: Examples of kernel functions (applied to cells and their corresponding vertices) with a varying tolerances towards the presented neighbour orderings. The variables xs and ys are vertex associated data representing x and y coordinates.
Note that our aim is to give an intuition as to why ordering may not matter sometimes. The fact that a mesh is really "ordering tolerant" hinges on the commutativity and associativity of operations such as addition, for which the latter may not hold for floating point representations.

Chapter 8

Exploiting structure

With the mesh data laid out conveniently in memory, alongside the meta-data express the detected structure, we discuss how the core-computation may be modified to use the structure which we have extracted.

8.1 Basic structured region parameters

Recall our skeleton example in figure 7.4b of a core-computation loop over structured and unstructured regions. Figure 8.1 shows the skeleton of a similar loop structure, except that the iteration is over the structured regions of a cell-vertex relation-map. While this loop faithfully shows the general loop structure, it glosses over much of the important detail.

8.2 Structured region boundaries

We first consider the meta-data we need to store for each structured region; the loop skeleton provides some hints at this, as it uses the interior_structured_rows and interior_structured_cols fields associated with each structured region.

Let us consider the example mesh from figure 7.9, which is modified in figure 8.2 to highlight the cell-vertex relation-map. We would like the structured

```
for region in structured_regions:
  for row in region.interior_structured_rows:
    for column in region.interior_structured_cols:
      # Compute cell_id and vertex_ids ... (details omitted)
      kernel_function(cell_id, vertex_ids)

# Loop over unstructured region (omitted)
```

Figure 8.1: Pseudo-code of a loop over structured regions with respect to a cell-vertex relation-map.

region loop to iterate over the interior structured elements (the lightly coloured cells in the example). Since all structured regions are fundamentally based on a vertex structured region, we parametrise the interior structured region on that basis. The following parameters allow us to specify the iteration space directly:

- The first structured *vertex* in the structured region (*first vertex*).
- The number of *vertex* rows[1] and columns in the structured region (*#rows* and *#cols*).
- The *vertex* offset in rows and columns from which the sub-rectangle of interior structured elements begin (*interior start row offset* and *interior start col offset*).
- The *vertex* offset in rows and columns to which the sub-rectangle of interior structured elements extends (*interior end row offset* and *interior end col offset*).
- The first structured element in the structured region (*first element*).

Based on these parameters we can write a loop over structured regions, as show in figure 8.3

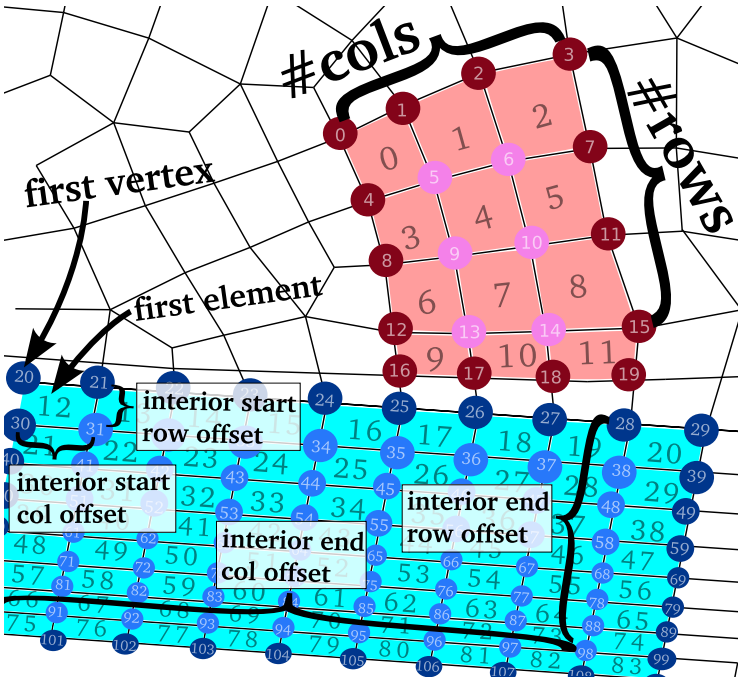
8.3 Applying neighbour reordering

In section 7.6 we discussed the problem of neighbour ordering, and proposed solutions to tackle the problem. Our chosen approach was to fix a single ordering for each structured region, and store it as meta-data. To apply this in our structured loop, we define a simple function (shown in figure 8.4) which takes an ordered sequence of elements, and a "compass" defining the ordering of the neighbours, and returns the reordered sequence of elements. The reordering function can then be used straightforwardly as shown in figure 8.5

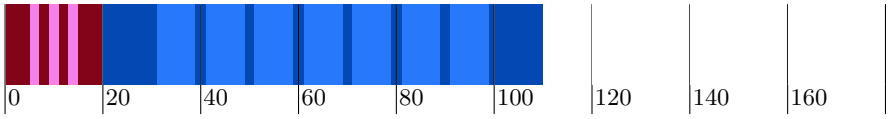
8.4 Chapter summary

In this chapter we covered how the core-computation loop can be transformed to *exploit* the detected structured regions by making use of the meta-data associated with them.

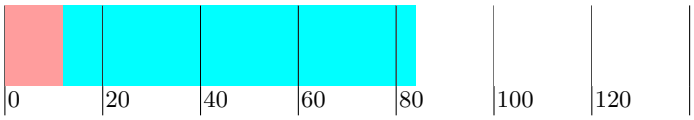
[1] Strictly speaking we may omit the number of vertex rows



(a) A mesh with two structured regions, coloured in blue and red. Vertices are indicated by circles. Lighter shades denote interior structured elements, and darker shades denote fringe structured elements.



(b) The vertex storage layout in memory. The numbers indicate vertex ids, and the colours correspond to the mesh diagram above.



(c) The cell storage layout in memory. The numbers indicate cell ids, and the colours correspond to the mesh diagram above.

Figure 8.2: A reproduced version of figure 7.9, annotated with structured region meta-data; the mesh depicts cell structured regions and their boundaries as induced by a cell-vertex relation-map.

```
for region in structured_regions:
  num_cell_cols = 1 + region.interior_end_col_offset - region.
    interior_end_start_offset
  num_vtx_cols = region.num_cols
  vtx_base = region.first_vertex

  for row in [region.interior_start_row_offset .. region.
    interior_end_row_offset]:
    for col in [region.interior_start_col_offset .. region.
    interior_end_col_offset]:
      cell_id = region.first_element + num_cell_cols*row + col

      vertex0_id = vtx_base + num_vtx_cols*row + col
      vertex1_id = vtx_base + num_vtx_cols*row + (col + 1)
      vertex2_id = vtx_base + num_vtx_cols*(row + 1) + col
      vertex3_id = vtx_base + num_vtx_cols*(row + 1) + (col + 1)

      vertex_ids = [vertex0_id, vertex1_id, vertex2_id, vertex3_id]


      kernel_function(cell_id, vertex_ids)

# Loop over unstructured region (omitted)
```

Figure 8.3: The loop from figure 8.1 expanded to show more detail, particularly of address calculation.

```
# elements is an array of four elements
# compass is a permutation of [0, 1, 2, 3]
def reorder_four_elements(elements, compass):
  e0 = elements[compass[0]]
  e1 = elements[compass[1]]
  e2 = elements[compass[2]]
  e3 = elements[compass[3]]

  return [e0, e1, e2, e3]
```

(a) A function to apply a reordering of four elements.

```
# elements is an array of n elements
# compass is a permutation of 0..n-1
def reorder_elements(elements, compass):
  new_elements = []

  for target_index in compass:
    elements.append(elements[target_index])

  return new_elements
```

(b) A generalised function to apply a reordering to an arbitrary number of elements.

Figure 8.4: Definition of functions that apply a reordering to mesh elements.

```
for region in structured_regions:
  num_cell_cols = 1 + region.interior_end_col_offset - region.
    interior_end_start_offset
  num_vtx_cols = region.num_cols
  vtx_base = region.first_vertex

  for row in [region.interior_start_row_offset .. region.
    interior_end_row_offset]:
    for col in [region.interior_start_col_offset .. region.
    interior_end_col_offset]:
      cell_id = region.first_element + num_cell_cols*row + col

      vertex0_id = vtx_base + num_vtx_cols*row + col
      vertex1_id = vtx_base + num_vtx_cols*row + (col + 1)
      vertex2_id = vtx_base + num_vtx_cols*(row + 1) + col
      vertex3_id = vtx_base + num_vtx_cols*(row + 1) + (col + 1)

      vertex_ids = [vertex0_id, vertex1_id, vertex2_id, vertex3_id]
      reordered_vertex_ids = reorder_four_elements(vertex_ids,
    region.compass)

      kernel_function(cell_id, reordered_vertex_ids)

# Loop over unstructured region (omitted)
```

Figure 8.5: The loop from figure 8.3 with reordering including.

Chapter 9

Implementation

We discuss our implementation of the detection algorithms discussed previously, as well as the core-computation for the airfoil computation.

9.1 Mesh storage format

All information about a particular mesh is stored in `*.p` and `*.p.part` files, which are formatted as ZIP archives. The various mesh components are stored in separate files within the archive.

9.1.1 `*.p.part` file

A `*.p.part` file stores the basic mesh information which do not related to structure. Unstructured core-computations (which only use the relation-maps) can be executed on these files. The contents are as follows:

- The number of elements in each entity set.
- Associated data, typically vertex-associated spatial coordinates.
- Relation-maps, for example an edge-cell relation-map.
- A vertex-vertex relation-map derived from existing relation-maps. This is used for structure detection later on.

9.1.2 `*.p` file

A `*.p` file is merely a `*.p.part` with additional structure-related data appended to it. The contents include the following:

- Structured region meta-data, as discussed in chapter 8,
- Renumbered associated data and relation-maps, as discussed in chapter 7

- Renumbering maps of the mesh entity sets, useful for debugging, visualisation, and analysis.

9.2 Building `*.p.part` files

The mesh builder is written in C++, and uses the Minizip library [30] for creating the archive and adding files to it. The individual components of the mesh, as described in subsection 9.1.1, are created in a custom format which uses Protocol Buffers, Google's open source serialization library [14].

`*.p.part` files can be built from two sources:

- `*.dat` files, a custom format generated by OP2 [19].

  Example: `bin/run-builder build-from naca0012.dat naca0012.p.part`

- `*.msh` files, generated by by Gmsh, an automatic 3D finite element mesh generator [13].

  Example: `bin/run-builder build-from-msh naca0012.msh naca0012.p.part`

9.3 Structure detection

Structure detection is implemented entirely using Python [11].

Python, amongst other things being an interpreted language, is inherently slow. In addition, it was targeted as a rapidly changing prototype implementation, and as such was not written with performance in mind. For this reason, we cannot fairly compare its performance to that of the core-computation, an application written in optimised C++ code.

Structure detection is split amongst three (self-evidently named) files:

- `detect_node_structure.py` defines the class DetectNodeStructure. It implements the detection of multiple structure vertex regions, using the class DetectQuadStructure defined in `quad_mesh.py` to detect individual regions.

- `detect_cell_structure.py` defines the class CellStructureFromNodeStructure , which detects overlaid cell structured regions given the detected vertex structured regions.

- `detect_edge_structure.py` defines the class EdgeStructureFromNodeStructure , which similarly detects overlaid edge structured regions given the detected vertex structured regions.

9.4 Building `*.p` files

A `*.p` file can be built using the functions defined in `write_structure_info.py`:

- The function write_structured_node_info writes to file the detected vertex structured regions and their associated set of relation-maps and associated data.
- The function write_structured_cell_info writes the respective cell structure region information.
- The function write_structured_edge_info writes the respective edge structure region information.

The ZIP archive files are read from and written to using the zipfile module in Python's standard library.

9.5 Running structure detection

There are two python scripts which can be used to run structure detection and generate a `*.p` file : a) `detect_and_append_structure.py` detects a single vertex region, overlaid with a cell structured region and an edge structured region. b) `detect_multiple_structure.py` detects multiple vertex regions by random seed-vertex sampling. Each is overlaid with a cell structured region and an edge structured region.

9.6 Using structure in core-computation

This is the code which facilitates the use of the detected structured regions.

9.6.1 Cells kernel

`cell_computation.h` defines a template function run_structured_cells , which can execute any kernel function over a cell-vertex relation-map.

```
template<typename InArgs, typename OutArgs>
static inline void run_structured_cells(
  const QuadCellMesh& mesh,
  void cell_kernel(const int cell_id, const QuadNeighbours&
    vertex_neighbours, const QuadCellMesh&, const InArgs&, const
    OutArgs&),
  const InArgs& inputs,
  const OutArgs& outputs);
```

The function takes the following arguments:

- a QuadCellMesh object, through which all mesh information, save for associated data, is accessed: relation-maps, structured region meta-data, and so on. This is forwarded along to the kernel function.
- a kernel function cell_kernel which takes as arguments a cell id, the ids of its vertices, the QuadCellMesh object, and other arguments to be explained momentarily.

- An argument of arbitrary type InArgs, which contains associated data with read-only access. This is forwarded along to the kernel function.
- An argument of arbitrary type OutArgs, which contains associated data with read-write access. This is forwarded along to the kernel function.

9.6.2 Edges kernel

`edge_computation.h` defines a template function run_structured_edges, which can execute any kernel function over edge-vertex and edge-cell relation-maps.

```
template<typename InArgs, typename OutArgs>
static inline void run_structured_edges(const QuadCellMesh& mesh,
  void edge_kernel(const int,const int, const int, const int, const
     QuadNeighbours&, const QuadNeighbours&, const int, const
    QuadCellMesh&, const InArgs&, const OutArgs&),
  const InArgs& inputs, const OutArgs& outputs)
```

The function takes similar arguments to run_structured_cells, only differing in the type of kernel function. The argument edge_kernel is a kernel function which takes as arguments:

- the ids of the two neighbouring vertices
- the ids of the two neighbouring cells
- the ids of the vertex neighbours of the aforementioned cells
- The forwarded arguments, as described above.

9.7 Airfoil computation

`run-airfoil-computation.cc` is a rewrite of the airfoil computation found in OP2 [20]. The computation is described in more detail in subsection 10.1.4.

9.8 Chapter summary

In this chapter we detailed important aspects of our prototype implementation, including the mesh storage format and third-party libraries used. We described the important function that have been implemented, and crucially the scope of our implementation within our general discussion of structure detection.

Chapter 10

Evaluation

In this chapter we evaluate various aspects of our work in structure detection. We first present a technical evaluation of the algorithm, assessing the quality of its output and runtime performance. We also look at the performance characteristics achieved when incorporating our structure detection into the core-computation work-flow.

10.1 Benchmarks

We briefly introduce the sample meshes used as benchmarks.

10.1.1 Airfoil grid mesh

This is a mesh generated using the `naca0012.m` script [19], following the NACA0012 definition for constructing an airfoil upper surface. It is parametrised by two variables I and J which denote the length and width of the airfoil boundary. The default airfoil mesh size is is built with $I = 400$ and $J = 600$, containing about 720,000 vertices. Topologically, the vast majority of the mesh forms a single large rectangular structure. Figure 10.1 shows a sample rendering.

10.1.2 NACA0012 mesh

This mesh was generated from `naca0012.geo`, kindly provided by George Ntemos, using Gmsh [13]. It represents the construction of a NACA0012 airfoil at zero incidence. The mesh is fairly coarse, containing roughly 15,000 vertices. Figure 10.2 shows sample renderings.

10.1.3 NACA0021 mesh

This mesh was generated from `naca0021.geo`, kindly provided by Harry Davis, using Gmsh [13]. It represents the construction of a NACA0021 airfoil at a 60

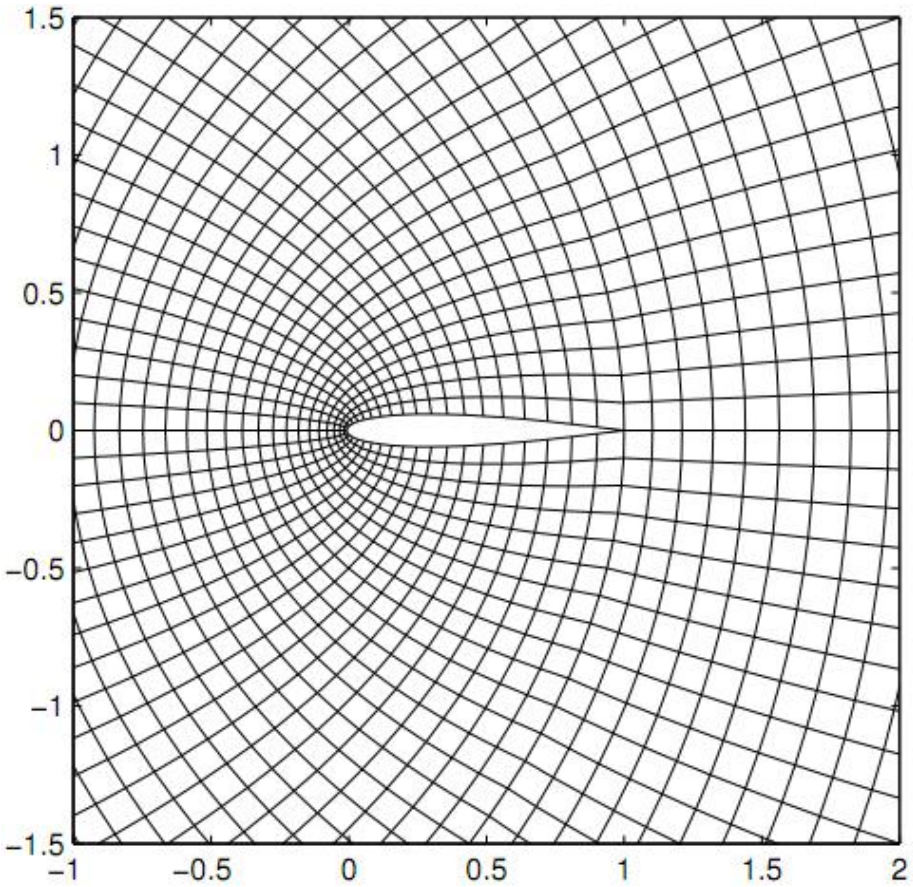
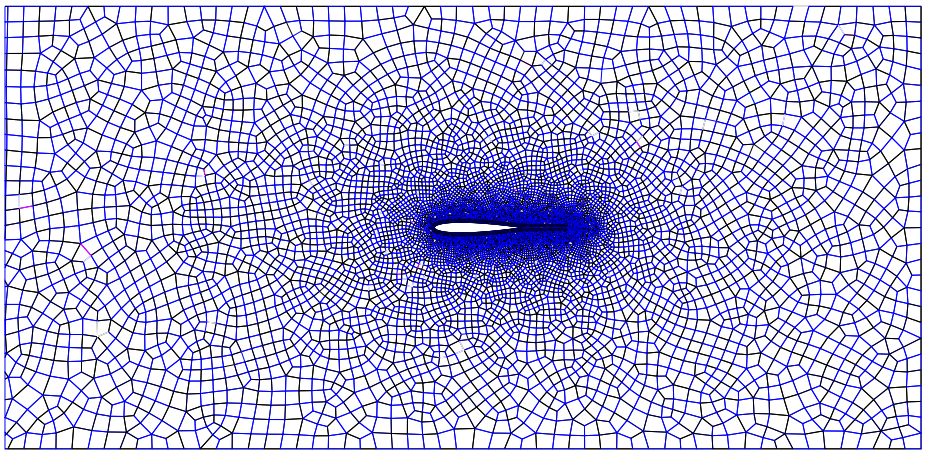
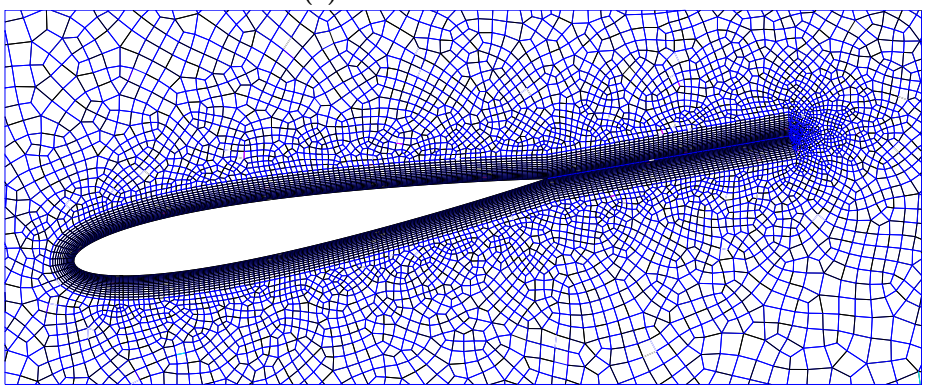


Figure 10.1: A rendering of a small sample of the airfoil mesh generated by [19]. The image was obtained from [28].

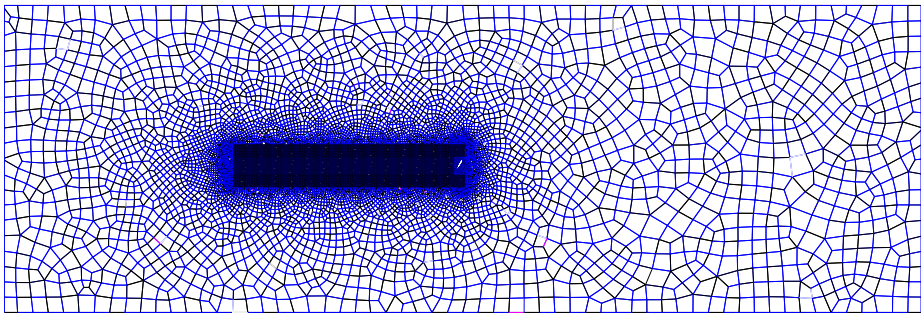


(a) A wide shot of the airfoil.

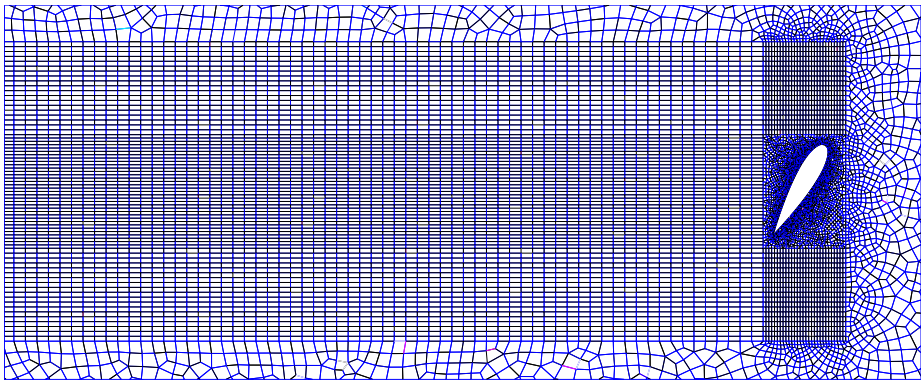


(b) A close-up at the airfoil border. Note the highly structured region formed directly around the airfoil.

Figure 10.2: A rendering of the naca0012 airfoil mesh.



(a) A (very) wide shot of the airfoil. The diagonal facing spec in the middle is the airfoil!



(b) A close-up at the airfoil border. Note the large structured region formed behind the airfoil.

Figure 10.3: A rendering of the naca0021 airfoil mesh.

degree angle of attack. The mesh contains roughly 20,000 vertices. Figure 10.3 shows sample renderings.

10.1.4 The airfoil computation

The airfoil computation is an example core-computation which is used as an example application in OP2 [20]. The computation applies the volume method to solve the 2D Euler equations iteratively, refining the solution until a steady state is reached. Within each iterations it performs multiple loops over cells and meshes, which involve both reading and writing data.

This application serves as a good benchmark, as it allows us to a) compare our performance against OP2's, and b) compare out computed results against OP2's, to ensure correctness.

10.2 Metrics

10.2.1 Core-computation metrics

There are three types of core-computation executions which we evaluate:

- Baseline OP2 execution time The runtime of the core-computation with OP2 [21], which does not use any structured region information. Mesh elements are accessed using indirection maps.
- Crystal execution time The runtime of the core-computation with Crystal using structured region information. Mesh elements in structured regions are accessed directly using address calculations; mesh elements in unstructured regions are accessed using indirection maps.
- Unstructured Crystal execution time The runtime of the core-computation with Crystal *without* providing structured region information. Mesh elements are accessed using indirection maps. The structured loops are still present in the code, though they are skipped at runtime due to missing structured region information. In effect, this is a competing implementation of the same iteration algorithm used by OP2.

All execution measurements exclude initialization and cleaning up time, which in any case have been found to be negligible.

10.2.2 Detection metrics

- Structured region size:

  This is the number of vertices found in a particular structured region. The general notion may refer to the average of such number.
- Number of structured regions detected
- Detection coverage: Either the number or percentage (as defined in context) of vertices in the mesh detected as structured elements.

10.3 Structure detection

The first step in evaluating our structure detection algorithms, manifested through our implementation, is to answer the simple question: does it work? Using detection metric here is useful, but at a fundamental level the best way to assess whether the implementation works as expected is to visualise the result.

10.3.1 Airfoil grid

Our first structure detection run is performed on the airfoil grid, as it is the simplest case for which we can assess correctness. Figure 10.4 shows the result of running structure detection on a very small version of the airfoil grid mesh.

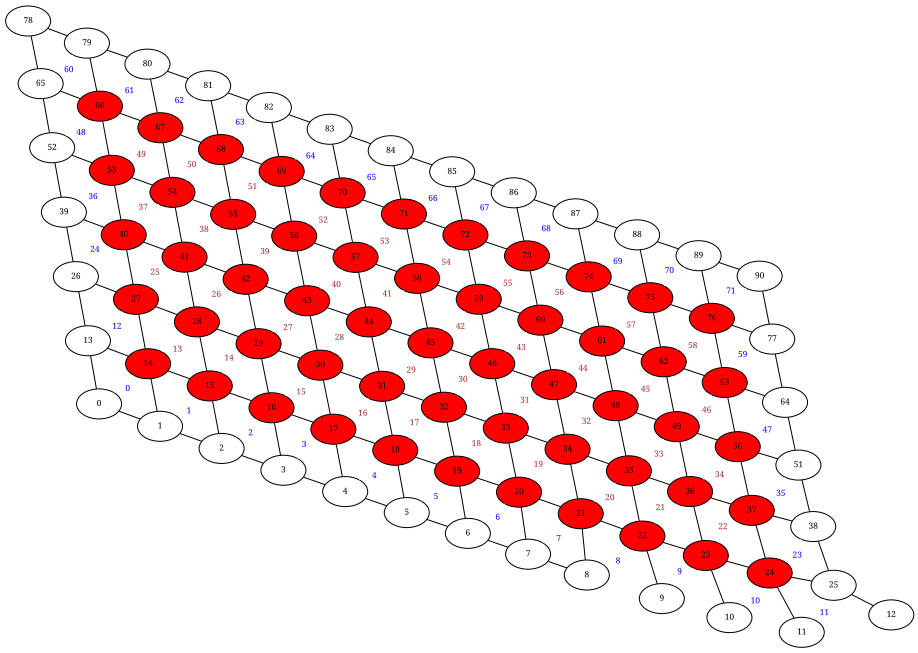


Figure 10.4: A rudimentary visualisation[1] of the detected structure in an airfoil grid mesh. The mesh is parametrised with $I = 4$ and $J = 6$, resulting in 91 vertices. Structured vertices and cells are denoting in bright and dull red, respectively. Note that the numbering shown denotes the old mesh numbering.

We can clearly see that the structured regions are as we would expect, forming a two-dimensional lattice which spans to include all structured vertices.

10.3.2 NACA0012

We now consider more complex meshes where more interesting patterns of structure can be found. First, we detect the structure in the NACA0012 mesh, depicted in figure 10.5. Notice how the highly structured region surrounding the borders of the airfoil is cleanly detected as one big structured region. However, looking at the small patches of structure, it is clear that there have been some losses. All these losses, however, are distinctly attributable to our choice of length-first search, which we described in subsection 5.3.1 as being eager. We shall see an example of this shortly.

Figure 10.6a shows the structured region frequency for the NACA0012, av-

[1] The custom `*.dat` used by the authors of this mesh does not make it amenable to visualisation by popular tools. This image was generated using a Python script outputting a graph description, with suitable colour and label information, which is then passed to the `dot` [8] utility to visualize.

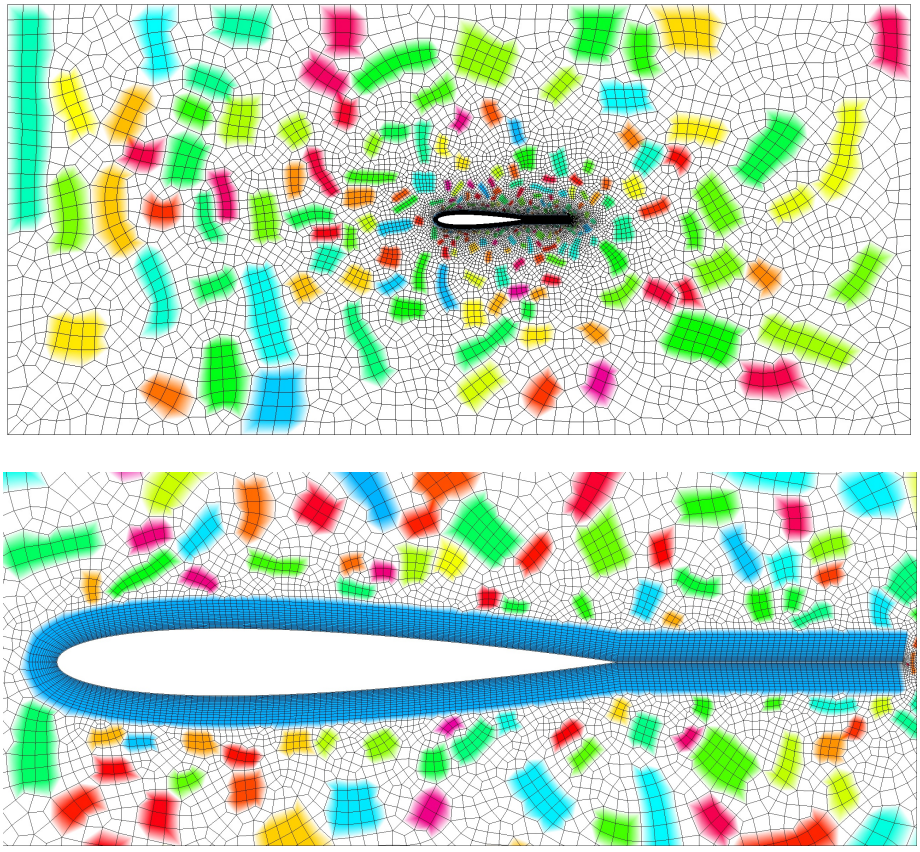
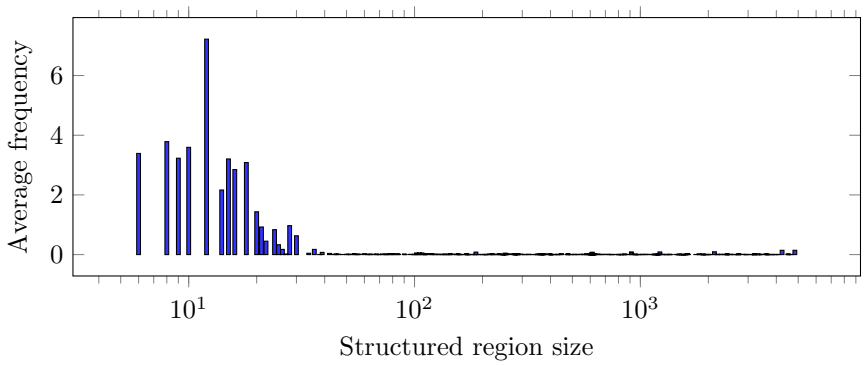


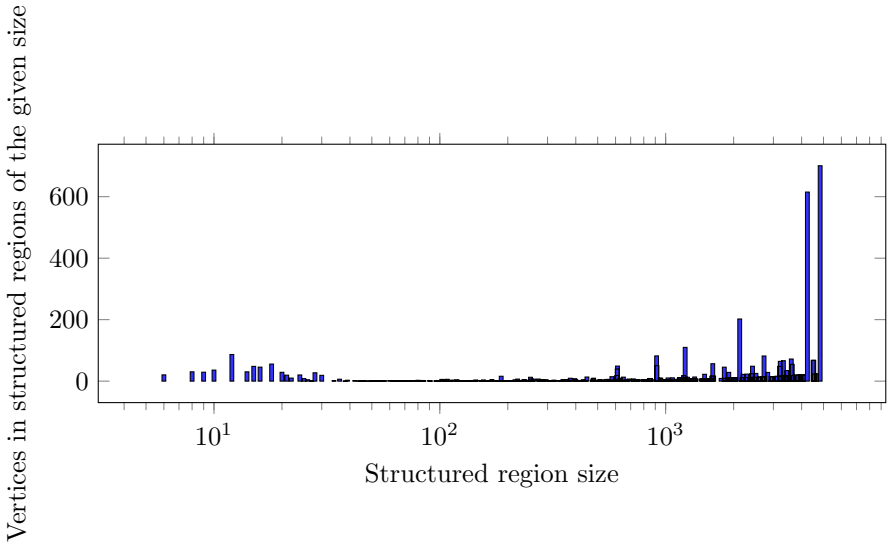
Figure 10.5: A visualisation of the detected structure in the NACA0012 mesh.

eraged over 200 runs. As would be intuitively expected, a small number of large structured regions are discovered, whereas small structured regions are abundant. Structured region size seems to follow a skewed normal distribution, with a peak value at 12.

Figure 10.6b on the other hand plots the frequency of vertices *belonging to* a structure of a given size. For example, if we partition a mesh into five regions of size 10 and one region of size 100, respectively, then 50 vertices in the mesh belong to a region of size 10, and 100 vertices belong to a region of size 100. Notice how the sizes of structured regions follow a bimodal distribution: they are typically either very small or very large.



(a) Plot of the distribution of structured region sizes.



(b) Plot of the distribution of structured region sizes to which vertices belong.

Figure 10.6: Statistics related to the structured region sizes of the NACA0012 mesh.

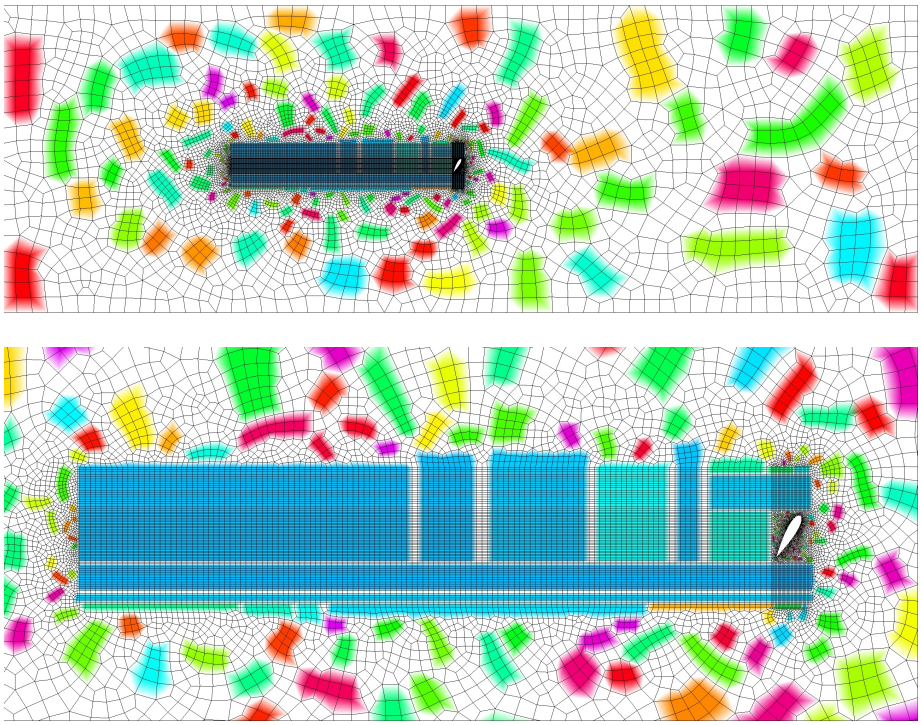


Figure 10.7: A visualisation of the detected structure in the NACA0021 mesh.

10.3.3 NACA0021

Next we look at the results of structure detection for the NACA0021 mesh, shown in figure 10.7. Interestingly the algorithm failed to detect the obvious targets as a single large chunk, and has instead fragmented it into a few relatively large blocks. Examining the boundaries of this large chunk, we find that it has many "traps", that is small portions where an eager structure detection algorithm may extend beyond where its supposed, and then get stuck in a sub-optimal configuration. Figure 10.8 shows the detected structure when we hand-pick a seed vertex which does not fall into such a "trap".

Figure 10.9a shows the structured region frequency for the NACA0021, averaged over 200 runs. The results are similar to the NACA0012 results in figure 10.6a, and in fact the same peak value of 12 is present here.

Figure 10.9b shows the frequency of vertices *belonging to* a structure of a given size. It is also similar to the corresponding plot for NACA0012 in figure 10.6b.

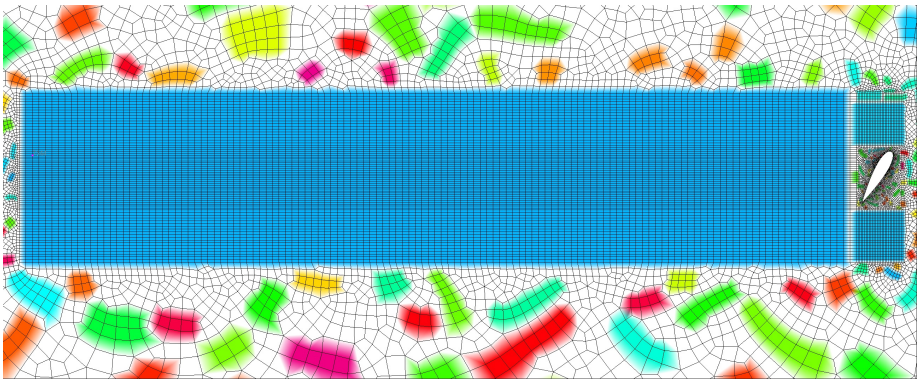


Figure 10.8: A visualisation of the detected structure in the NACA0021 mesh when hand-picking a "good" seed vertex.

10.4 Core-computation runtime performance

10.4.1 Experimental method

All experiments are performed on a model HP 800 G1 TWR. The processor is an Intel® Core™ i7-4770 3.40GHz with four physical cores[2], supporting the SSE 4.1/4.2 and AVX 2.0 instruction sets [5]. The main memory capacity of the machine is 16GB.

The creation of the source mesh, where applicable, is never timed, and Crystal loads and uses the data in-memory as standard arrays. Creation refers to storage of the relevant entity set meta-data (number of vertices cells, etc), relation maps, and the associated data (typically spatial coordinates). This non-expensive process involves converting the mesh into a custom data format based on Protocol Buffers [14] with ZIP compression applied using the Minizip library [30]. Our justification behind this is to offer a simplified programming model for data retrieval, manipulation and storage. To our best knowledge this presents no significant advantage to Crystal, if at all.

The Crystal detection code is written in Python and run using the Python interpreter version 2.7.4 [11].

The Crystal execution code is compiled using the Intel® C and C++ Compilers version 14.0.2 20140120 [4] with the following optimization flags set:

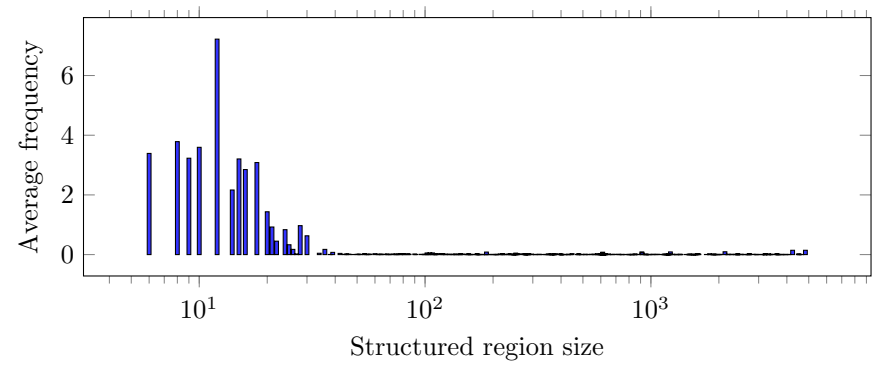
- -xHost

  Generate instructions for the highest instruction set and processor available on the compilation host machine.

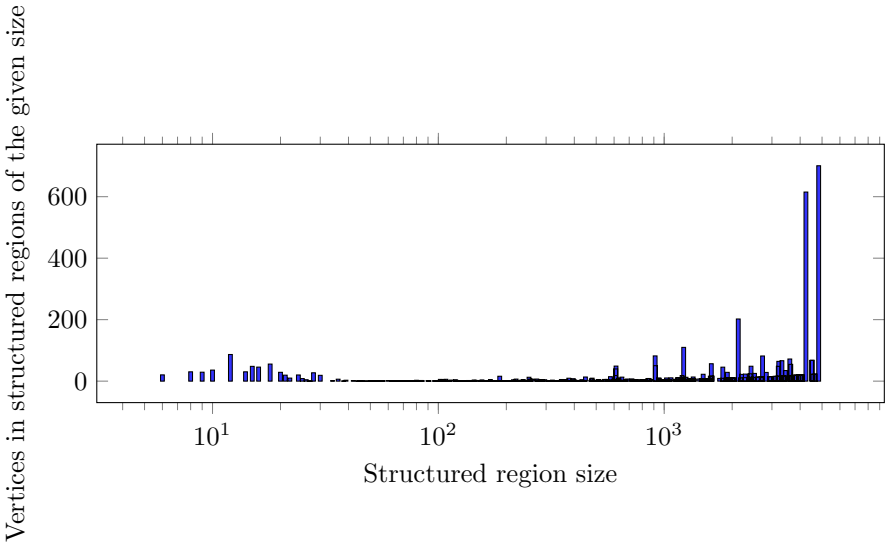
- -static-intel

  Link Intel provided libraries statically.

[2]Note that the experiments were run serially



(a) Plot of the distribution of structured region sizes.



(b) Plot of the distribution of structured region sizes to which vertices belong.

Figure 10.9: Statistics related to the structured region sizes of the NACA0021 mesh.

- -ipo

  Enable multi-file IP [inter-procedural] optimization between files.

- -O3

  Optimize for maximum speed and enable more aggressive optimizations that may not improve performance on some programs.

The baseline OP2 execution code is compiled using the same version of Intel® C and C++ Compilers using the provided Makefile. We use the sequential CPU-only version (double-precision airfoil_plain) [20], compiled by issuing the command `make airfoil_seq`. The optimization flags set are as follows:

- -O3 (as above)

- -xAVX

  May generate Intel® Advanced Vector Extensions (Intel® AVX), Intel® SSE4.2, SSE4.1, SSSE3, SSE3, SSE2, and SSE instructions for Intel® processors.

- -parallel

  Enable the auto-parallelizer to generate multi-threaded code for loops that can be safely executed in parallel.

10.4.2 Impact of input mesh numbering

Results

Crystal versus Baseline

The performance of Crystal, using structure information or otherwise, is far superior to that of the baseline OP2 run. This difference can be attributed to runtime overheads present in the OP2 implementation, notably the *staging* of data[3].

Impact of numbering on Crystal with structure information

The performance of Crystal, given structure information, is agnostic to the original mesh numbering. This is because the vast majority (99.5%) of vertices are detected to be part of a single monolithic structured region, which Crystal then renumbers in any case.

[3]Staging is the process of transferring data to a temporary holding location for further processing. Staging is usually performed piecemeal on several blocks of data.

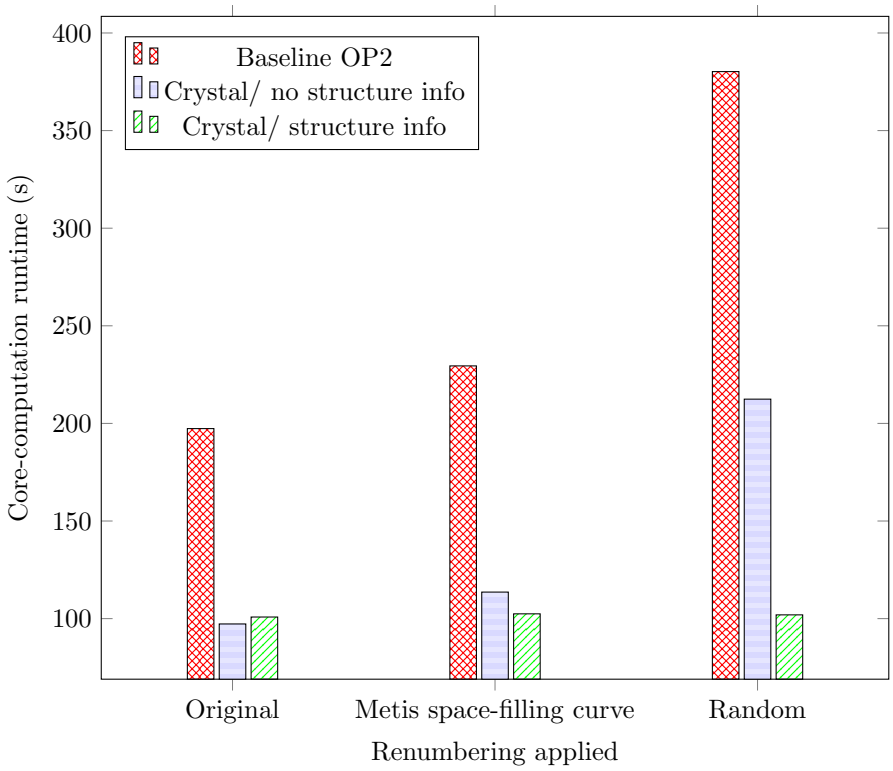


Figure 10.10: the Impact of numbering on various types of executions.

Impact of numbering on Crystal without structure information

The airfoil grid is almost fully structured *by construction*, and has a very good numbering of its elements. Crystal's structure detection then cannot improve on cache locality, merely introducing a small performance overhead. When different numberings are applied to the mesh, however, Crystal using structure information outperforms its non-structure-based counterpart by a modest 11% improvement with the METIS numbering, and over twice as fast with random numbering.

As a general discussion point, following several investigations into our implementation, we have determined that vectorisation techniques are failing to apply. We suspect that performing further work on optimising the code can make it amenable to vectorisation, and hence superior in performance even when the original numbering is a good one.

10.5 Detected structure quality

It is difficult to produce a single definitive metric with which we can unequivocally define the quality of our detected structure. We can, however, look at properties that are in line with that target, and derive useful met. We can, however, look at properties that are in line with that target, and derive useful metrics by combining them.

Percentage of mesh elements detected as structured: This metric is most useful when we know the maximum value that would be expected or possible. Without this knowledge, the metric is a one-sided test: a high percentage is a positive sign, whereas a lower percentage is indeterminate.

Structured region size: Larger structured regions are desirable, but similarly to the previous, the maximum possible size is mesh-dependant and may not be known.

Number of structured regions: On its own this metric is a tricky one to resolve. It is useful, however, to act as a relative scale for other metrics such as structured region size.

Chapter 11

Conclusion

We have covered in much depth the topic of structure in meshes:

- The meaning of structure and strategies for extracting it.
- Exposing the detected structure through the mesh representation.
- Best exploiting structure in computation.

11.1 Extracting structure

In chapter 4 we toy with the idea of structure in meshes. We intuitively derive definitions for structure and corresponding suitable data structures. Building on the insights which we have reached, we discuss in chapter 5 the abstract properties of structure detection algorithms, and develop sketches of a few such algorithms. In chapter 6 we take one of these algorithms, length-first search, and refine its description into a detailed formal algorithmic description.

11.2 Exposing structure

Having *identified* structured regions , in chapter 7 we discuss methods of renumbering the mesh elements to expose the structure efficiently, and of including auxiliary structure-related meta-data, all while maintaining exactly the topology and geometry of the original mesh. We also discuss detecting structured regions based on the available relation-maps, overlaying structured regions of other mesh elements, and how to best represent them.

11.3 Exploiting structure

Moving forward in chapter 8 we uncover the final piece of the puzzle. We present how we can *leverage* and exploit the exposed structure in computations by transforming the data accesses passed to the computation kernel.

11.4 Trailing off

We briefly discuss our implementation in chapter 9, including details about its general organization, the mesh data format used, as well as the major functions developed. Finally in chapter 10, we evaluate our work in terms of the effectiveness of the structure detection itself as well as the performance results it achieved.

Chapter 12

Future works

This project covered extensive grounds on the topic identifying and leveraging structured regions in quad meshes. With that said, there are plenty of interesting future directions to be pursued, and intriguing tangents to be followed.

12.1 Non-quad meshes

In their current state, our algorithms only work on meshes with exclusively quadrilateral faces. Future works may accommodate for quadrilateral-based structure in mixed element-type meshes, for example a combination of quadrilaterals and triangles, is quite straightforward. Extending our work to detect structures composed of a different element-type would be a natural extension of this.

A more challenging direction would be the detection of arbitrary topological patterns, perhaps those which can be inscribed within a quadrilateral. As a side-path, we tried building a small prototype of a structure detection algorithm which takes as input a pattern inscribed in a quadrilateral, and then automatically detects lattice formations of that pattern. It would be very interesting to follow through with this idea and see whether it has any potential.

12.2 Different types of structure

We only consider a particular kind of structure, a discrete Cartesian space, for its simplicity and suitability for data storage and high performance processing. There may, however, be other topological structures to be found and exploited. Utilising hierarchical-based structures, the topics of [17] and [2], seems to be a promising direction.

12.3 3D meshes

Our work deals with quadrilaterals, typically modelling surfaces of three-dimensional objects. One may ponder whether similar structural properties, and associated benefits, exist for meshes modelling the volumes of three-dimensional objects, in other words meshes composed of three-dimensional elements such as tetrahedra

An interesting direction to investigate is developing a structure detection algorithm which is given a cube with a certain pattern inscribed within it, extending the concept we discussed above to embedding of an arbitrary vertex topology within a cube. A defining question to answer would be whether structure in a tetrahedral mesh can be represented as the tiling of cubes in three-dimensional space.

12.4 Investigate structure growth strategies

In chapter 5 we discuss various structure detection strategies, in particular those for growing a rectangular region, in addition to more general structure detection strategies. Various inspirations can also be drawn from [10] and [9], notably the methods of particle shooting.

One may wish to go further, incorporating use of geometric information into the structure detection process, as have [18] and [24].

12.5 Structure detection in parallel

Our structure detection algorithm was designed to run in a serial fashion, sequentially choosing random seed points and growing structure from there. Developing a parallel variant of the algorithm is important for scalability, as some meshes may simply be too large to process on a single machine. The discussion on non-contiguous detection and merging structured blobs is a good starting point for this. Particle shooting methods as described in [10] and [9] may also be of interest here.

12.6 Structure detection for parallel computation

As discussed in related works, some of our objectives for efficient exploitation of structure conflicted with those of works such as [2] and [17], as they were oriented towards exploiting parallelism rather than improving computational runtime. There may indeed be much untapped[1] potential in parallelism which can be attained by adapting our methods. Even if this were not the case, the sizes of some meshes necessitate parallel computation, as a full mesh becomes much too large for a single machine to manage.

[1] On our part, that is.

12.7 Geometry based detection

We have mentioned at the start that our approach is purely topological, completely ignoring any geometric information. This at odds with some other techniques such as graph partitioning based on space-filling curves [23], as well techniques which refine, and hence modify, the mesh geometry [2]. Samet offers an interesting overview of space-filling curves in this regard [25].

A future pursuit may choose to abandon this separation, exercising a union of geometry and topology as an aid for structure detection. Though applied to specific *geometric* problems, the techniques used by [24] and [18] may serve as useful inspiration.

12.8 Adaptive runtime detection

Thus far we have considered structure detection as a pre-processing stage to the core-computation; an ambitious work may strive to develop an online version of the detection algorithm. The structure detection would be performed in parallel to the core-computation, with structured regions incrementally hot-swapped in between iterations as they are detected.

A variant may incorporate the detection algorithm as part of a work dispatch strategy for parallel computations. In [17] for instance, the mesh is adaptively refined at runtime as part of a dynamic load balancing framework; the method we discuss, however, would not modify the mesh itself, but rather its partitioning.

Chapter 13

Bibliography

[1] Ira H Abbott and Albert E Von Doenhoff. *Theory of wing sections: including a summary of airfoil data*. Courier Dover Publications, 2012.

[2] Benjamin Karl Bergen and Frank Hülsemann. Hierarchical hybrid grids: data structures and core algorithms for multigrid. *Numerical linear algebra with applications*, 11(2-3):279–291, 2004. URL: `http://onlinelibrary.wiley.com/doi/10.1002/nla.382/pdf`.

[3] The Boeing Company. What is an airfoil? URL: `http://www.boeing.com/companyoffices/aboutus/wonder_of_flight/airfoil.html`.

[4] Intel Corporation. Intel® c and c++ compilers. URL: `https://software.intel.com/en-us/c-compilers`.

[5] Intel Corporation. Intel® Core™ i7-4770 processor (8m cache, up to 3.90 ghz). URL: `http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz`.

[6] Karen Daniels, Victor Milenkovic, and Dan Roth. Finding the largest area axis-parallel rectangle in a polygon. *Computational Geometry*, 7(1):125–148, 1997. URL: `http://cogcomp.cs.illinois.edu/papers/maaprJ.pdf`.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. URL: `http://dl.acm.org/citation.cfm?id=1327492`.

[8] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C North, and Gordon Woodhull. Graphvizopen source graph drawing tools. In *Graph Drawing*, pages 483–484. Springer, 2002.

[9] David Eppstein, Michael T Goodrich, Ethan Kim, and Rasmus Tamstorf. Approximate topological matching of quadrilateral meshes. In *Shape Modeling and Applications, 2008. SMI 2008. IEEE International Conference on*,

pages 83–92. IEEE, 2008. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4547954.

[10] David Eppstein, Michael T Goodrich, Ethan Kim, and Rasmus Tamstorf. Motorcycle graphs: canonical quad mesh partitioning. In *Computer Graphics Forum*, volume 27, pages 1477–1486. Wiley Online Library, 2008. URL: www.ics.uci.edu/~goodrich/pubs/geomproc.pdf.

[11] Python Software Foundation. Python version 2.7.4. URL: https://www.python.org/download/releases/2.7.4.

[12] Matteo Frigo, Charles E Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *Foundations of Computer Science, 1999. 40th Annual Symposium on*, pages 285–297. IEEE, 1999. URL: http://supertech.csail.mit.edu/papers/Prokop99.pdf.

[13] Christophe Geuzaine and Jean-Franois Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre-and post-processing facilities, 2008.

[14] Google. Protocol buffers. URL: https://developers.google.com/protocol-buffers/.

[15] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library. *Version 1.0, Dept. of Computer Science, University of Minnesota*, 1997.

[16] Arnold Martin Kuethe and Chuen-Yen Chow. *Foundations of aerodynamics: bases of aerodynamic design*. Wiley New York, 1986.

[17] Xiaolin Li and Manish Parashar. Hierarchical partitioning techniques for structured adaptive mesh refinement applications. *The Journal of Supercomputing*, 28(3):265–278, 2004. URL: nsfcac.rutgers.edu/TASSL/Papers/icpp02_hpa.pdf.

[18] Jonathan E Makem, Cecil G Armstrong, and Trevor T Robinson. Automatic decomposition and efficient semi-structured meshing of complex solids. In *Proceedings of the 20th international meshing roundtable*, pages 199–215. Springer, 2012. URL: www.imr.sandia.gov/papers/imr20/Makem.pdf.

[19] OP2. naca0012 .m mesh generator, 2012. URL: https://github.com/OP2/OP2-Common/blob/master/apps/mesh_generators/naca0012.m.

[20] OP2. Op2 airfoil, 2014. URL: https://github.com/OP2/OP2-Common/tree/master/apps/c/airfoil/airfoil_plain.

[21] OP2. Op2-common, 2014. URL: https://github.com/OP2/OP2-Common.

[22] Daniel A Reed, Loyce M Adams, and Merrell L Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *Computers, IEEE Transactions on*, 100(7):845–858, 1987. URL: `http://ieeexplore.ieee.org/ielx5/12/35266/01676980.pdf`.

[23] Phil Ridley. Guide to partitioning unstructured meshes for parallel computing. Technical report, HECToR: UK National Supercomputing Service - Numerical Algorithms Group, April 2010. URL: `www.hector.ac.uk/cse/reports/unstructured_partitioning.pdf`.

[24] Luigi Rocca, Nikolas De Giorgis, Daniele Panozzo, and Enrico Puppo. Fast neighborhood search on polygonal meshes. In *Eurographics Italian Chapter Conference 2011*, pages 15–21. The Eurographics Association, 2011. URL: `http://www.inf.ethz.ch/personal/dpanozzo/papers/EGIT11-RocDeGPanPup.pdf`.

[25] Hanan Samet. *Foundations of multidimensional and metric data structures*. Morgan Kaufmann, 2006.

[26] Yuan Tang, Rezaul Alam Chowdhury, Bradley C Kuszmaul, Chi-Keung Luk, and Charles E Leiserson. The pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, pages 117–128. ACM, 2011. URL: `http://people.csail.mit.edu/yuantang/pochoir_spaa11.pdf`.

[27] Timothy J Tautges. Moab-sd: Integrated structured and unstructured mesh representation. *Engineering with Computers*, 20(3):286–293, 2004.

[28] Kyrylo Tkachov. Accelerating unstructured mesh computations using custom streaming architectures. Master's thesis, Imperial College London, 2012.

[29] Henk Kaarle Versteeg and Weeratunge Malalasekera. *An introduction to computational fluid dynamics: the finite volume method*. Pearson Education, 2007.

[30] Gilles Vollant. Minizip: Zip and unzip additionnal [sic] library. URL: `http://www.winimage.com/zLibDll/minizip.html`.