Optimizing pointer linked data structures

Holm, C.W.M.

Chapter 3

Theory of Grids

Data structure transformations still need a substantial research effort to make them more effective. In this chapter, possible future directions for empowering data structure transformations are explored. Although not directly implementable, the techniques and methods may very well be so in the future when further research has been conducted in this area. The feasibility of restructuring data structures is explored by treating the restructuring and analysis of the linked data structures as a graph problem. In essence we try to redefine the topology of the data structure used in such a program into a more optimal data structure, given the actual structure of the graph. For example, if a graph representing the data structure has a Hamiltonian path, it is possible to restructure that part of the data structure to an array.

In general, we will not consider control flow in the analysis, as for instance is done in the work on shape analysis [19], but only analysis of the topology of the graph. As such this is the first step in handling the restructuring problem and the methods need to be extended with control flow analysis in the future. An important result in this chapter is that we show that control flow analysis, or other additional knowledge, is required in order to effectively identify sparse grids and other regular structures, although it is feasible to apply direct approaches on complete grids. This result is the guiding principle that led to the later chapters in this thesis.

The shape of a pointer linked data structure cannot be determined from type information only (pointers point to arbitrary objects). For example, a structure type containing two recursively typed pointers in each object, may represent a doubly linked list, a tree, a grid, or any other kind of graph formable by two outgoing arcs per vertex (and by edge decomposition, many other kinds

of graphs may be formed, including complete graphs).

Special optimizations may be carried out on for example lists, trees and grids in some cases. For example, a list is linearizable (i.e. can be transformed into an array), grids are two dimensional and can be transformed into two dimensional arrays (using for example a jagged diagonal layout). In order to carry out a linearization of a grid it must be known by the compiler or the run time that the structure actually is a grid and not for example a doubly linked list or a tree.

In these cases it is important that there exists a theoretical base in order to choose the right methods when doing such optimizations or related analysis.

This chapter introduces the problem named MINIMUM CONFINED COMPONENTS, which is related to finding *rows* or *columns* in graphs. MINIMUM CONFINED COMPONENTS is shown to be NP-complete. However, for special classes of graphs such as complete grids and trees, the MINIMUM CONFINED COMPONENTS problem can be solved in polynomial time, and for many other graphs, we can use a heuristic method for decomposing a graph into confined components. There are several areas where confined components have both theoretical and practical applications. We give an algorithm capable of decomposing a graph into confined components in $O\left(|V|^2 + |V||E|\right)$ time. This algorithm is shown to be optimal for trees, complete grids and complete triangular grids.

We introduce the concept of orthogonal sets of confined components that can be used to formally describe grid styled graphs. For many cases it turns out that we do not need to compute the orthogonal sets but can instead make a good guess that can be proven to be orthogonal using a simple but efficient algorithm. We introduce such an algorithm that is also mostly parallel, with a sequential complexity of $O\left(L * (|V| + |E|)\right)$, where L is the number of dimensions in the grid.

Several practical applications are discussed in detail, giving pointer based code that could be optimized based on the theory introduced in this chapter. A compiler could allow for loop permutations on pointer linked structures (i.e. iterating row by row instead of column by column) by identifying orthogonal sets of confined components, and possibly also the application of complex loop analyses such as the polytope model that can assist with skewing and parallelization [39] of loops.

Section 3.1 gives definitions used later on in the chapter. Though some of the definitions are not new, they are given in order to make the chapter self contained. Confined components are introduced and the MINIMUM CONFINED COMPONENTS problem is proved to be NP-complete in Section 3.2. In Section 3.3 *CCDA*, a near optimal polynomial time algorithm capable of finding

confined components is introduced and discussed. Section 3.4 introduces orthogonal edge labeling, a concept that builds on confined components. Finally, theoretical and practical applications of the confined components are discussed in Sections 3.5 and 3.6.

3.1 Definitions

This chapter introduces new graph theoretic concepts, some of which are tightly related to grids. In this section we introduce the fundamentals that are needed to step by step build up the notion of confined components and the formal definition of grids. Note that for the remainder of this chapter, we are dealing with directed graphs unless otherwise specified. A component $G_C(V_C, A_C)$ as used here, is a subset of an existing digraph $G(V, A)$ s.t. $V_C \subseteq V \wedge A_C \subseteq A$. Note that the symbol $\veebar$ may be used as the symbol for *exclusive or*.

Definition 1. Unilaterally Connected Component (UCC)

A unilaterally connected component [7] is a subset of nodes and arcs such that for every pair of nodes p, q there is a path from p to q or from q to p (or both).

Definition 2. Exclusive Unilateral Component (EUC)

An exclusively unilateral component is a subset of nodes and arcs such that for every pair of nodes p, q there is a path from p to q if and only if there is not a path from q to p.

Definition 3. Traceable Component

A traceable component is a subset of nodes, $V_C \subseteq V$ and arcs $A_C \subseteq A$ that has a Hamiltonian path, i.e., there is a permutation of V_C, say $v_1, v_2, \ldots, v_r$ with $(v_i, v_{i+1}) \in A_C$ for $1 \leq i < r$, where $r \leq |V|$.

For grids, there exists an assignment of arc colors and numerical vertex labels such that in for instance the two-dimensional case, each vertex has two labels L_1 and L_2 (representing *row* and *column* indices) and each arc is assigned one of two colors C_1 and C_2 so the color represents *row* or *column* directions. We can define this more exact as follows:

Definition 4. A digraph $G(V, A)$ with vertices V and arcs A is called a grid if there exist an arc coloring C_1 and C_2 and a vertex labeling L_1 and L_2, such that:

- There is a maximum of two incoming arcs per vertex.
- All incoming arcs for a vertex $v \in V$ have different colors.
- There is a maximum of two outgoing arcs per vertex.
- All outgoing arcs for a vertex $v \in V$ have different colors.

and that for all $(u, v) \in A$ we have:

$$\begin{cases} L_1(u) < L_1(v) \wedge L_2(u) = L_2(v) & \textbf{if } (u,v) \text{ has color } C_1 \\ L_1(u) = L_1(v) \wedge L_2(u) < L_2(v) & \textbf{if } (u,v) \text{ has color } C_2 \end{cases} \tag{3.1}$$

In essence, this means that the labeling L_1 and L_2 take on the role of row and column index.

When we refer to grids in this chapter we may use the terms complete, sparse and directed grid in order to avoid ambiguity in the interpretation from the reader. A complete grid is a grid where the labeling relation is $L_n(u) = L_n(v) - 1$ instead of $L_n(u) < L_n(v)$. A sparse directed grid is what has been defined in Definition 4, and an example can be seen in Figure 3.1. An undirected grid is a directed grid where the arcs have been replaced with edges.

In the literature, the notion of grid is also often used for undirected graphs where vertices have a two dimensional index in $\{1, \ldots, n\} \times \{1, \ldots, n\}$ for some n, and vertices are adjacent when in one dimension, they have the same index, and in the other dimension, their index differs exactly one. If we direct all edges in such a graph to the right or down, we obtain a complete grid.

It is easy to build a sparse grid from a complete grid, by removing vertices and replacing the arcs to and from that vertex with arcs that maintain the same row and column connectivity.

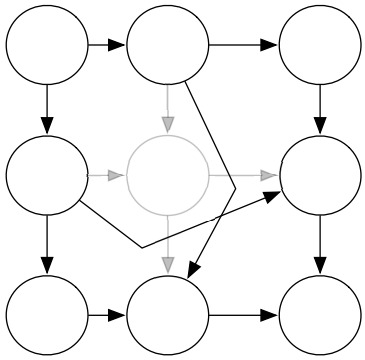

Figure 3.1: A Sparse Grid. As can be seen, the middle vertex of the complete grid is removed. A sparse grid is formed by replacing the removed element with arcs while maintaining the row and column connectivity of the surrounding vertices.

Definition 5. Directed Feedback Vertex Set

Given a digraph $G(V, A)$, where V is the set of vertices and A the set of arcs. A subset $W \subseteq V$ is called a feedback vertex set, if each cycle in G contains at least one vertex of W.

3.2 Confined Components

Confined components is an important new concept, that is related to directed grids. The exact relationship will be explored in Section 3.4. For now, we can summarize them as offering a graph theoretic approach that let us identify rows and columns in a grid structure, without resorting to vertex labeling or arc coloring.

When we refer to contraction, we specifically refer to edge contraction, the mechanisms of which are described in detail in for example [57]. Edge contraction is commutative, so we can say that contraction of a whole component is the application of the edge contraction operation on every arc within the component.

Terminology wise, after a set of components have been contracted, the resulting graph is in line with strongly connected component theory called a *condensate*, where each vertex represents a component in the original digraph.

Definition 6. Confined Unilateral Component

A confined unilateral component is a traceable component C of a *directed acyclic graph* (DAG) D, so that if the component is contracted, there are no induced cycles in the digraph D' resulting from the contraction of C.

Proposition 7. *A Confined Unilateral Component C is Exclusively Unilateral*

Proof. A confined unilateral component C has no cycles since it is part of a DAG. As C is traceable, by definition it contains an Hamiltonian path. Thus for every pair of vertices $(u, v) \in C$, $u \neq v$ in the component there is a path from u to v or from v to u, since there is a Hamiltonian path, but not both since there are no cycles. Hence, the confined component is an exclusively unilateral component as given in Definition 2. □

Lemma 8. *A decomposition of a DAG into a set of confined unilateral components will have a topological order*

Proof. Since the decomposition consists of confined unilateral components, there are no cycles in the condensate of these components. Since there are no cycles, the condensed graph is a DAG and all DAGs have topological orderings[1]. □

Let the labeling from Definition 4 correspond columns and rows, we call the label for columns L_C and the labeling for rows L_R. Let $L_C^{\max}$ be the maximum value of the labels for all $L_C(v)$ where $v \in V$ and $L_R^{\max}$ the maximum value of the labels for all $L_R(v)$, and $L_C^{\min}$ and $L_R^{\max}$ the corresponding minimum labels.

The vertices whose label $L_C(v) = L_C^{\max}$ are in the right most column, $L_C(v) = L_C^{\min}$ in the left most column, $L_R(v) = L_R^{\min}$ in the top row and $L_R(v) = L_R^{\max}$ in the bottom row.

A vertex v is considered to be above u if $L_R(v) < L_R(u)$ and below u if $L_R(v) > L_R(u)$. A vertex v is considered to be left of u if $L_C(v) < L_C(u)$ and right of u if $L_C(v) > L_C(u)$.

Proposition 9. *For a finite 2D sparse or complete directed grid $G(V, A)$, the set of components formed by contracting either all the rows or all the columns form confined components.*

[1]The fact that DAGs have topological orderings is well known and is discussed in many text books on graph theory and algorithms such as for example [21], which contains a proof for this on page 362.

Proof. Since the graph is finite, there must be a left and right-most column. The right-most column must form a confined component as the column has no vertices on the right hand side, and no further vertices above the top most vertex, or below the bottom vertex (in the same column), thus it only has incoming arcs, and therefore it is not part of any cycle. If the column is removed, a new column is made the right-most column, the previously removed column cannot form a cycle with the current one (since no arcs are going back to the rest of the grid), and must be traceable or it would not be a column in a grid. Since none of the columns then form cycles with other columns and all columns are traceable, the columns meet the definition of confined components. The same reasoning holds when replacing columns with rows. □

The following theorem was proven by *Hans L. Bodlaender*:

Theorem 10. *Decomposing a DAG into the minimum number of confined components is NP-complete.*

Proof. The problem clearly belongs to NP. To show it is NP-hard, we use a transformation from Directed Feedback Vertex Set. Let an instance of Directed Feedback Vertex Set be given, i.e., a directed graph $G = (V, E)$ and an integer L. Build a directed acyclic graph $H = (W, A)$ as follows. For each vertex $v \in V$, we take two vertices x_v and y_v, and an arc (x_v, y_v). For each arc $a = (v, w) \in E$, we take three vertices, $z_{a,1}$, $z_{a,2}$ and $z_{a,3}$. We add arcs: $(z_{a,1}, z_{a,2})$, $(z_{a,2}, z_{a,3})$, $(x_v, z_{a,2})$, $(z_{a,2}, y_w)$. Set $K = |V| + |E| + L$.

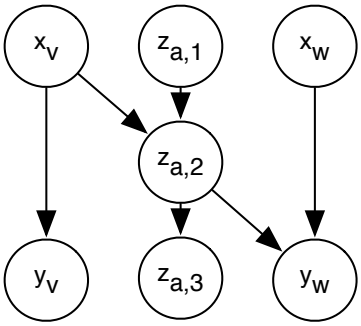

Figure 3.2: Construction: subgraph for arc $a = (v, w) \in E$

Note that H is acyclic: taking first all vertices of the form x_v, then all vertices of the form $z_{a,1}$, then all vertices of the form $z_{a,2}$, then all of the form $z_{a,3}$, and finally all vertices of the form y_v gives a topological order.

Claim. G has a feedback vertex set of at most L vertices if and only if H has a confined decomposition with at most K vertex sets.

Proof. $\Rightarrow$: Suppose G has a feedback vertex set $X \subseteq V$ with $|X| \leq L$.

Now, for each arc $a \in E$, we take the vertex set $\{z_{a,1}, z_{a,2}, z_{a,3}\}$. We say that this set represents a. For each vertex $v \in X$, we take two vertex sets, each with one element: $\{x_v\}$, and $\{y_v\}$. For each vertex $v \in V - X$, we take one vertex set with elements $\{x_v, y_v\}$. We say that this set represents v.

Clearly, each of these at most $|E| + |V| + L$ vertex sets induce a subgraph of G that has a Hamiltonian path.

We need to show that the graph obtained by contracting each vertex set to a single vertex is acyclic. First, note that for all $v \in X$, the vertex set $\{x_v\}$ has no incoming arc, so can never be on a cycle. Similarly, for $v \in X$, the set $\{y_v\}$ has no outgoing arc, so is not on a cycle. Now only consider the other vertices in the graph obtained by contracting vertex sets. A vertex that is a contracted set representing a vertex has only arcs to and from vertices that is a contracted set representing an arc, and moreover, if we have an arc from the set representing $v \in V - X$ to the set representing $a \in E$, then v is the head of a. Similarly, if we have an arc from the set representing $a \in E$ to the set representing $v \in V - X$, then v is the tail of that arc. Thus, if we have a cycle in the graph obtained by contraction, then this directly corresponds to a cycle in $G[V - X]$ which contradicts the fact that X is a feedback vertex set.

$\Leftarrow$: Suppose H has a confined decomposition with at most K vertex sets.

We say a confined decomposition is *fine,* if for each arc $a \in E$, we have a set $\{z_{a,1}, z_{a,2}, z_{a,3}\}$. We claim that there also is a fine confined decomposition with at most K vertex sets. We can obtain such a fine confined decomposition, by repeating the following steps:

- Consider a vertex set V_i that contains $z_{a,2}$ for some $a \in E$ but that does not contain $z_{a,1}$. Then, note that $\{z_{a,1}\}$ forms a one-element vertex set in the decomposition, as the vertex $z_{a,1}$ only has an arc to $z_{a,2}$ and no incoming arcs. If V_i contains an element of the form x_v, then we replace the sets V_i and $\{z_{a,1}\}$ by the sets $\{x_v\}$ and $V_i - \{y_v\} \cup \{z_{a,1}\}$. Otherwise we replace the sets V_i and $\{z_{a,1}\}$ by the set $V_i \cup \{z_{a,1}\}$.

- Consider a vertex set V_i that contains $z_{a,2}$ for some $a \in E$ but that does not contain $z_{a,3}$. Then, note that $\{z_{a,3}\}$ forms a one-element vertex set

in the decomposition, as the vertex $z_{a,3}$ only has an arc from $z_{a,2}$ and no outgoing arcs. If V_i contains an element of the form y_v, then we replace the sets V_i and $\{z_{a,3}\}$ by the sets $\{y_v\}$ and $V_i - \{y_v\} \cup \{z_{a,3}\}$. Otherwise we replace the sets V_i and $\{z_{a,3}\}$ by the set $V_i \cup \{z_{a,3}\}$.

We repeat the steps above while possible, and will obtain a fine confined decomposition with the same number or fewer vertex sets.

For a fine confined decomposition, vertex sets that contain vertices of the form x_v or y_v can be of the following types: there is a vertex $v \in V$ with the set of the form $\{x_v\}$, the set of the form $\{y_v\}$, or the set of the form $\{x_v, y_v\}$. Let $X \subseteq V$ be the vertices in V with a set of the form $\{x_v\}$ in the fine confined decomposition. Clearly, for $v \in X$, also the set $\{y_v\}$ is in the decomposition. So the number of sets in the fine confined decomposition equals $2\,|X| + (|V| - |X|) + |A|$, and hence $|X| \leq L$.

We will now show that X is a feedback vertex set in G. Suppose not. Then we have a cycle in $G\,[V - X]$, say with successive vertices $v_0, v_1, v_2 \ldots v_{r-1}, v_r = v0$. Now, for each i, $0 \leq i < r$, the vertex set $\{x_{v_i}, y_{v_i}\}$ has an arc in the set $\{x_{v_{i+1}}, y_{v_{i+1}}\}$. And thus, these vertex sets form a cycle in the contracted graph, which is a contradiction.

So, X is a feedback vertex set in G of size at most L. □

As H can be constructed in polynomial time from G, and as Directed Feedback Vertex Set [29] is NP-complete, the theorem follows. □

Apart from its application context, Theorem 3.2 is also interesting if we compare it with some classic results in algorithmic graph theory. Compare the notion of unilaterally connected components with strongly components: a strongly connected component in a digraph is a set of nodes such that for every pair p, q, there is a path from p to q and from q to p. Partitioning a digraph in its strongly connected component can be done in linear time with a well known application of depth first search (see e.g., [11, Chapter 22]. Many problems on directed acyclic graphs are efficiently solvable (e.g., [11, Chapter 24.2]. Determining if a digraph has a Hamiltonian path is known to be NP-complete [29], but is trivially solvable in linear time for directed acyclic graphs.

Although the Minimum Confined Components problem is NP-complete, it is relatively easy to decompose a DAG into confined components, where the number of confined components is not necessarily minimal.

3.3 Confined Components Decomposition Algorithm

We have devised an efficient greedy polynomial time algorithm that can decompose a DAG into confined components. The method devised is illustrated as Algorithm 1. This algorithm is optimal in terms of finding the minimum number of components, for trees, complete square and triangular grids; and works as expected on cyclic digraphs, i.e. it does not explode in complexity or incorrectly place nodes that are in a cycle in a component.

The algorithm starts at any source node and finds the heaviest path without joins (nodes with in-degree > 1). As metric for weight, we use the sum of the out-degree of all the branching nodes (nodes with out-degree > 1) in the component. Though one could use the number of nodes, the argument for using the sum of the branch node out-degree is that by maximizing the number of out-edges from a selected component, the in-degree sum in the rest of the graph is minimized when removing the component, thereby reducing the number of join nodes in the rest of the graph. Although not empirically evaluated here, this approach also turned out to work better in practice for the example graphs that the algorithm was tested on.

Only branches and sequences of nodes are placed in the path. Since no merge nodes are in these paths, the search space will be restricted to a tree. The heaviest path can be found with a DFS based heaviest path search, if a heavier path is found, the current heaviest path will be updated (when the search has reached a leaf). Though, the copy of the current path to the heaviest path does not look linear in the first case, it can easily be made so by implementing an incremental copy scheme. This could work by first allocating the extra space, and then copying backwards, stopping with the copy when the two paths join together. Thus, even though there may be a copy in every leaf, this copy will only copy nodes that have not yet been copied. So, the copy complexity in total is only $O(|V|)$.

When the heaviest path has been extracted, the nodes in that path are removed from the graph and any new source nodes (due to the node removal) are added to the list of sources and the computed heaviest path without merges is added to the list of confined components.

The algorithm also works in the reverse direction, by starting in sink nodes and constructing the components but switching merge nodes with branch nodes.

Note that the algorithm runs in polynomial time, this is easy to see as the algorithm perform DFS traversals and removes at least one node from the

Algorithm 1 Confined Component Detection Algorithm (CCDA)

```
1  def heaviestNonMergingPath(v, currentPath, heaviestPath):
2    # Stop at joins so we are bounded to a tree
3    if len(v.sources) > 1:
4      return
5    currentPath.push(v) # Add node to current path
6    # Increment the length metric for branches
7    if v.targets > 1:
8      currentPath.passedBranches ++
9    # Visit all targeted nodes
10   for tgt in v.targets:
11     heaviestNonMergingPath(tgt, currentPath, heaviestPath)
12   # Check if the path is longer than the known
13   # longest path
14   if currentPath.passedBranches
15     > heaviestPath.passedBranches
16   or heaviestPath.isEmpty():
17     heaviestPath = currentPath
18   if v.targets > 1:
19     currentPath.passedBranches --
20   currentPath.pop()
21
22 def FindConfinedComponents(G):
23   cc = []
24   sources = G.findAllSourceNodes()
25   for src in sources:
26     path = []
27     heaviestNonMergingPath(src, [], path)
28     cc += [path]
29     G.removeNodes(path)
30     sources += G.findNewSources()
31   return cc
```

graph for each DFS.

Theorem 11. *CCDA is correct and results in a set of confined unilateral components.*

Proof. There are two properties to be proven. Firstly, that each component will form traceable paths. Secondly, that the contracted graph will be cycle free. Let $G(V, A)$ be a DAG. Let $In(v)$ and $Out(v)$ be the in- and out-degrees of the vertex v. Let C_k be a component in the graph G and let us call the contracted graph $G_C = (V_C, A_C)$, where V_C is the contracted components and A_C is the arcs between the components.

Claim. Each component when selected is traceable.

Proof. First, let us rewrite the `heaviestNonMergingPath` function more formally as follows:

$$f(v,p) = \begin{cases} p & \textbf{if } In(v) > 1 \\ p \oplus v & \textbf{if } In(v) \leq 1 \wedge Out(v) = 0 \\ L(f(w, p \oplus v) | \forall (v,w) \in A & \textbf{if } In(v) \leq 1 \wedge Out(v) > 0 \end{cases} \tag{3.2}$$

where $v \in V$ is a vertex in G, p is a sequence of vertices from G, $\oplus$ is concatenation, and L selects the heaviest sequence of the list of sequence parameters.

The property to prove is that the result of $f(v, p)$ yields a traceable path (sequence of vertices).

The initial invocation of `FindConfinedComponents` invokes $f(v, \emptyset)$, with $In(v) = 0$. Then we have two cases: $Out(v) = 0$ or $Out(v) > 0$. If $Out(v) = 0$, the result will be $\emptyset \oplus v = v$, a single vertex and therefor traceable. If $Out(v) > 0$, all subsequent invocations will be $f(w, \emptyset \oplus v)$ or $f(w, v)$, or in other words have as argument a traceable path, together with vertex w, who is directly connected to the last node of that path.

Now assume f is invoked by f, with as argument a traceable path p and a node v, for which an edge exists in A connecting the last node in p with v. Then in case $In(v) > 1$, the result is p and therefore traceable. In case $In(v) \leq 1 \wedge Out(v) = 0$, the result is $p \oplus v$, and therefore traceable. In case $In(v) \leq 1 \wedge Out(v) > 0$, all invocations of $f(w, p \oplus v)$ have as argument a traceable path. The proof follows by induction. □

Claim. There will be no cycles in the contracted graph.

Proof. The second property follows directly from the fact that as seen above, any invokation of f is either with a vertex v whose indegree is 0, or with a path p whose last node has an edge to v. In case there would be a cycle, there would be an arc from the rest of the graph to a node v in the contracted component. This can only be the case if there would be an invokation of $f(p, v)$, with $In(v) \geq 2$. But in this case, v would not have been added to the contracted component. □

Since both the claims are valid the theorem holds. □

Theorem 12. *CCDA finishes in polynomial time.*

Proof. CCDA works by doing a DFS bounded by the tree formed by the cut-set of all $In(v) > 1$, this DFS search is $O(|V| + |E|)$. Whenever a DFS is complete, at least the source node is removed from the graph and placed in a component. Thus CCDA has a complexity of $O\left(|V|^2 + |V||E|\right)$. □

Theorem 13. *For trees, CCDA finds the minimum number of confined components and is therefore optimal.*

Proof. This is easy to prove, we first show that a decomposition of a tree into confined components must have at least as many components as there are leafs, and then that the algorithm will generate one component per leaf.

Claim. Decomposition into confined components will have at least as many components as there are leafs.

Proof. Assume that this would not be the case, then there would be at least one confined component where there were two leafs. However, such a decomposition is clearly not traceable and contradicts the definition of a confined component. □

Claim. CCDA generates exactly one component per leaf.

Proof. Since we are dealing with a tree, the algorithm will start with the root of the tree. The algorithm then performs a DFS, locating a traceable sequence that ends in a node that has out-degree 0 or in-degree > 1, however, since the graph in question is a tree, there are no vertices with in-degree > 1. Hence, the last node in the extracted component will be a leaf, but in order to reach that vertex, the algorithm will only descend through vertices that are not leafs. Hence, when starting the find longest non merging path algorithm from the root of a tree, the extracted component will contain exactly one leaf vertex.

When extracting one component, it is removed from the tree; this will not introduce any new leafs in the new graph, as a component will be removed starting with the tree's root, and not at any internal vertex.

The above reasoning applies to the further application of the subtrees formed by the component removal. □

Since a covering set of confined components must contain at least as many confined components as there are leafs in the tree, and the algorithm will generate exactly one confined component per leaf, the theorem follows. □

Theorem 14. *For complete (directed) grid-graphs, CCDA finds the minimum number of confined components and is therefore optimal.*

Proof. Consider an $m \times n$ grid where all the downward pointing arcs are labeled "*down*" and rightward pointing arcs are labeled "*right*", if there is no such labeling of the grid, then rotate it so that this labeling is valid.

Claim. Any trace in a grid must be a combination of down and right steps

Proof. Follows by definition of $m \times n$ grids. □

Claim. A confined component trace can consist out of arcs, either labeled right or down, but not a combination of these two.

Proof. Assume that this is not the case, then we have a (partial) trace p, q, r, where (p, q) is labeled *down* and (q, r) labeled *right*. In this case, there must be a vertex s, which is connected as follows: (p, s) labeled right, (s, r) labeled down. In this case, there is a path from p, q, r to s, and a path from s to p, q, r. These two paths however form a cycle between the two components contradicting the definition of a confined component. □

Claim. A confined component will have at most $\max(m, n)$ vertices in it.

Proof. This is trivial as a confined component will not contain arcs of different labels, the only components that can be constructed form parts of a row or a column in the grid. □

Claim. The algorithm will result in a set of components, all of length $\max(m, n)$.

Proof. Since the graph is a grid, the algorithm starts in the upper left corner of the grid (the only vertex whose in-degree is 0). The algorithm will then locate the longest non-merging path, this must either be the upper row or the left-most column. Assume without loss of generality that $m < n$, then the algorithm will find a row since each row is longer than each column. When this row of size n is removed, the grid shrinks to the size $\tilde{m} \times n$, with $\tilde{m} < m < n$. Further iterations will remove additional rows of length n and this happen while $\tilde{m} > 0$, until there are no vertices left to remove. □

Since CCDA will remove $\min(m, n)$ components of length $\max(m, n)$, the theorem follows. □

Theorem 15. *For complete triangular grids*[2]*, CCDA finds the minimum number of confined components and is therefore optimal.*

Proof. Follows in an analogous way to the proof of Theorem 14. What needs to be shown additionally, is that for the complete triangular grid there will never be a confined component including a diagonal arc. Assume this is not the case and the grid block has vertices p, q, r and s (named in clockwise order), and arcs (p, q); (p, r); (p, s); (q, r) and (s, r), where the diagonal arc is (p, r). From the component built by including (p, r), there are now paths through both q and s forming cycles with the component including (p, r), in turn contradicting the definition of a confined component.

Note that q and s cannot be part of the confined component at the same time (otherwise it is not traceable), and neither p, q, r nor p, s, r can form components (as shown in Theorem 14). Therefore, in the contraction of the graph, there will be at least two cycles because q and s will be contained in two different components. This contradicts Definition 6 and the theorem follows. □

Conjecture 16. *Finding the rows or columns in a given sparse grid is NP-complete*

Motivation: In some cases the set of rows or the set of columns in a sparse grid will be the minimum set of confined components. Therefore, finding these rows or columns is similar to finding the minimum number of confined components and hence most likely to be NP-complete.
As it is difficult to prove statically on a program that a data structure will not have any cycles, it is of interest to explore the behavior of the algorithm even on cyclic digraphs. We show here that for cyclic graphs, the algorithm will terminate in polynomial time and that any node that is in a cycle will not be placed in a component. The latter can be used to reject a graph for being cyclic during the runtime of a program.

Theorem 17. *Even for cyclic digraphs, CCDA is polynomial.*

Proof. As is the case for DAGs, at every outer loop pass, the algorithm will take one source node (in-degree = 0) and remove at least this node. The algorithm performs a DFS which is $O(|V| + |E|)$ in time, this DFS will terminate even in the presence of a cycle because a vertex in a cycle will be seen as a merging point in the current path. If there are no source node left in the graph (i.e.

[2]Triangular grids are defined in a similar way to complete square grids where each square is being tessellated into two triangles and the diagonal arc is pointing down.

the remaining nodes are parts of cycles), findMoreSources will return an empty list and the loop will thus terminate. Consequently, the complexity on cyclic DAGs is at most $O(|V|^2 + |V||E|)$. □

Theorem 18. *For cyclic digraphs, the CCDA will not embed nodes that are part of cycles in any component.*

Proof. We prove this by induction.

Claim. The first removed chain will not consist of nodes that are part of a cycle.

Proof. The algorithm will start at some source node (in-degree = 0), and remove a component without incoming arcs. A chain of nodes without incoming arcs cannot be part of a cycle. If this would be the case, then there would be an incoming arc to one of the nodes making that node's in-degree > 1. This contradicts the non-merging component property. □

Claim. A chain removed, assuming no nodes within a cycle have been removed before, will never be part of a cycle.

Proof. Consider a cycle, all the nodes on a cycles will initially have an in-degree > 0. The outer loop of the algorithm starts at a node whose in-degree is 0, and finds a chain of nodes with in-degree = 1. This chain of nodes cannot be part of a cycle. If this would be the case, one of the nodes would have an in-degree of at least 2, but this contradicts the non-merging component property. □

Since neither the first nor any subsequently removed components have nodes that are parts in a cycle, the theorem follows. □

3.4 Orthogonality

In this section we describe how confined components are used to define the notion of orthogonality for graphs. In fact this notion will allow us to infer grid-structures on any graph. Thereupon, this property can be used to transform a random graph traversal into a n-dimensional "grid" traversal. By doing so, these graph traversals can be transformed to n-dimensional loop structures (see Section 3.5). Orthogonality is based on labeling the arcs in the graph. All the arcs in the graph with identical labels are in turn grouped into the same arc set.

Definition 19. Orthogonal Graph
Let $G(V, E_1, E_2)$ be a digraph with vertex set V and arc sets E_1 and E_2. G is orthogonal iff $\forall p, q \in V : p \rightsquigarrow_{E_1} q \Rightarrow \neg(p \rightsquigarrow_{E_2} q)$. [3]

In general, this simply means that E_1 and E_2 are independent arc sets; it does for example not exclude branches or joins within the same arc set. Note also that if $G(V, E_1, E_2)$ is orthogonal, then by contraposition $G(V, E_2, E_1)$ is also orthogonal.

For discussing the results of the remainder of this section, we use the following notions. Let C_1, C_2 be two edge-disjoint vertex-covering confined component sets. Let E_1 be the set of edges in C_1 and E_2 the set of edges in C_2. Number all nodes in V according to their component order in C_2. Note that such an ordering exists as the contracted graph of the C_2 confined components is a DAG. Let $n(x)$ yield the number assigned to the vertex x. Let $f(x) = y$ such that $x, y \in E_2$. Let $out(x)$ be the out-degree of x with respect to E_2, that is for the vertex x, the number of out arcs in E_2.

Proposition 20. *Two edge-disjoint vertex-covering confined component sets C_1 and C_2 are orthogonal.*

Proof. Proof by contradiction: Assume the proposition does not hold, then we have $p \rightsquigarrow_{E_1} q$ and $p \rightsquigarrow_{E_2} q$. Contracting the set C_1 results in a singleton cycle formed by the edges of E_2, which contradicts the confined component assumption given in Definition 6. □

Lemma 21. *Two edge-disjoint vertex-covering confined component sets are ordered, that is:*

$$\forall\,(x, y) \in E_1 : n(x) < n(y) \tag{3.3}$$

$$\forall x, y \in \{v \in V : out(v) > 0\} : \begin{cases} n(x) < n(y) \Rightarrow n(f(x)) < n(f(y)) \\ n(x) = n(y) \Rightarrow n(f(x)) = n(f(y)) \end{cases} \tag{3.4}$$

Proof. Assume not, then either:
$\exists\,(x, y) \in E_1 : n(x) > n(y)$. Then because of edge disjointness of C_1 and C_2, (x, y) will be an edge between two different components in C_2. This clearly violates the ordering from the contracted C_2.

[3] With the notation of $p \rightsquigarrow_E q$ we mean that there exists a path from p to q in arc set E.

$\exists\,(x,y) \in E_1 : n(x) = n(y)$. Indicating that despite having a direct path in E_1 from x to y, there is a confined component in C_2 that includes this edge, contradicting the edge disjointness of E_1 and E_2.

$\exists x, y \in V : n(x) < n(y) \Rightarrow n(f(x)) \geq n(f(y))$. Then, clearly x and y are in different components of C_2, and either there is a cycle in the contracted graph of C_2 which is in contradiction with the definition of confined components, or $n(f(x)) = n(f(y))$. In the latter case there are edges in E_2 that are not part of any of the components in C_2, contradicting the definition of E_2.

$\exists x, y \in V : n(x) = n(y) \Rightarrow n(f(x)) \neq n(f(y))$. Then, there are edges in E_2 that are not part of any of the components in C_2, contradicting the definition of E_2. □

Definition 22. Strictly Ordered Orthogonality

We call two edge-disjoint vertex-covering confined component sets *strictly ordered orthogonal*.

Theorem 23. *The set of rows and the set of columns in a (sparse) directed grid are strictly ordered orthogonal.*

Proof. From Proposition 9 we know that the set of rows and the set of columns are confined component sets. Clearly these are also vertex covering.

Assume they are not edge disjoint, i.e. a an arc u, v is both in a row and a column. In that case there will be two colors assigned to the same arc which contradicts Equation 3.1. □

Theorem 24. *Let C_1 and C_1 be strictly ordered orthogonal, edge covering and let all the components have an Eulerian trail that is the same as the trace, then C_1 and C_2, may be represented as a (sparse) grid where the component sets C_1 is the set of rows and C_2 the set of columns (or columns and rows respectively).*

Proof. Consider Equations 3.3 and 3.4 from Lemma 21, and apply it on both sets. Let $m(x)$ be the number assigned to each vertex from its order in the contracted components of C_1. Then:

$$\forall (x,y) \in E_1 : n(x) < n(y) \wedge m(x) = m(y) \tag{3.5}$$

$$\forall (x,y) \in E_2 : m(x) < m(y) \wedge n(x) = n(y) \tag{3.6}$$

Note that, $<$ follows from the theorem and $=$ from the premise, as vertices involving the arcs of E_2 will be in the same component of C_2.

Assign a color C_1 to the arcs in E_1, and the color C_2 to the arcs in E_2. Then we have one color per arc, or $\forall(u,v) \in E : C_1(u,v) \veebar C_2(u,v)$. We can then rewrite Equations 3.5 and 3.6 as:

$$\begin{cases} n(u) < n(v) \wedge m(u) = m(v) & \textbf{if } C_1(u,v) \\ m(u) < m(v) \wedge n(u) = n(v) & \textbf{if } C_2(u,v) \end{cases} \tag{3.7}$$

which is equivalent to Equation 3.1.

Because the components have intra-component Eulerian trails that is identical to the intra-component traces, there can be at most one intra-component outgoing arc per vertex and at most one intra-component incoming arc per vertex. Each vertex can have at most one incoming and one outgoing arc of color C_1 and at most one incoming and one outgoing arc of color C_2. □

We provide a simple algorithm capable of verifying whether a decomposition is strictly ordered orthogonal. If such an algorithm is going to successfully run, it needs to ensure the following attributes.

- The graph must be shown to be initially cycle-free.
- For each component set, the components must be confined.
- For each component set, the components must be edge disjoint from the components in the other component sets.
- For each component set, the components must be vertex covering.

Algorithm 2 details the straight forward way to accomplish the verification.

Algorithm 2 Strictly Ordered Ortho. Verification Algorithm (SOOVA)

```
1  def HasCycles(G):
2    SCCs = Tarjan(G):
3    if length(SCCs) == 0:
4      return False
5    return True
6
7  def Contract(G, cs):
8    G2 = Graph()
9    Map = {}
10   for comp in cs.components:
11     v = Vertex()
12     G2.vertices += [v]
13     for v2 in comp.vertices:
14       Map[v2] = comp
15   for e in G.edges:
16     if Map[e.src] != Map[e.dst]:
17       G2.edges += [Edge(Map[e.src], Map[e.dst])]
18   return G2
19 # Given graph G and set of decompositions CC
20 # True iff the decomp is strictly ordered orthogonal
21 def IsDecompStrictlyOrderedOrtho(G, CC):
22   if HasCycles(G):
23     return False
24   for cs in CC:
25     cg = Contract(cs)
26     if HasCycles(cg):
27       return False
28     for e in cs.edges: # Check if edge disjoint
29       if e.tag == None:
30         e.tag = cs
31       else:
32         return False
33     for v in G.vertices: # Check if vertex covering
34       if v not in cs.vertices:
35         return False
36   return True
```

For Algorithm 2, the following two theorems can be proved in a straight forward manner:

Theorem 25. *The SOOVA algorithm is correct for* simple digraphs[4]*.*

Proof. In order to prove this, we need to show that the algorithm identifies a graph to adhere to the strictly ordered orthogonality property.

Claim. The algorithm detects cycles in the full data structure.

Proof. At the very start of the algorithm, Tarjan's SCC algorithm [48] is executed on the whole graph. Tarjan's SCC algorithm finds strongly connected components. There exists strongly connected components in a graph iff there are cycles in the same graph. Hence, the claim is valid. □

Claim. The algorithm detects that each component set is confined.

Proof. By contracting each component set, and finding cycles in the contracted graph, the premises of Definition 6 holds. □

Claim. The algorithm finds shared edges between the components and proves edge disjointness.

Proof. For all the edges in a component set, the algorithm will check if the edge has a tag (it is assumed that no tags have been set when the algorithm starts). If no tag is set a tag will be set for the next outer iteration and component set check. Since all edges in previous component sets are tagged when the next component set is checked, any edge in multiple component sets will be detected. □

Claim. The algorithm detects that each component set is vertex covering.

Proof. By iterating over every vertex, and checking if it is part of the current component set, the claim holds. □

Since all the claims are valid, the theorem follows. □

Theorem 26. *Given L component sets, verifying that a digraph is strictly ordered orthogonal takes $O(L * (|V| + |E|))$ time.*

[4]It should be trivial to device code that iterates over all edges, finding and eliminating multi edges and loops

Proof. The main algorithm has two top level items, a cyclicity check known to be $O(|V| + |E|)$ and a loop over the L component sets. The loop contains the following steps: *contract*, *cyclicity check*, an *iteration over all edges* and an *iteration over all vertices*. Contraction in turn is $O(C + |E|)$, where C is the number of components. However, a component has at least one vertex, so $O(C) \leq O(|V|)$. Cyclicity check is obviously $O(C + E')$ where E' is the number of edges between the components, worst case there are no edges within the components, and $O(E') = O(|E|)$. The remainder is the iteration over all edges $O(|E|)$ and the iteration over all vertices $O(|V|)$.

Summing the complexities together yields the following complexity $O((|V|+|E|) + L * (C + |E| + C + E' + |E| + |V|)) = O(L * (|V| + |E|))$ □

The algorithm is mostly parallel, we discuss the issues with parallelizing the algorithm informally here. While, being inherently a depth first search problem, for cycle detection there are some parallelizable algorithms (for example [14]). For this algorithm, the cycle detection will be the main bottleneck as the remaining steps are relatively easy to parallelize.

Edge disjointness checks are completely parallel and each edge can be checked concurrently under the assumption that the visitation check and the tagging are atomic together. This can for example be accomplished using either locks, transactional memories or *test and set* instructions. Parallel behavior here is $O(|E|/P)$ in the best case, but as the test and check must be serialized and if every single edge is shared with all sets, the complexity will instead be $O(L * ((E')/P))$ where E' is the number of edges per set. However, if this test fails the algorithm can immediately report that there are overlapping edges, so in practice it should just signal the failure and return, so the complexity is still $O(|E|/P)$.

Contraction of the components is also essentially parallel, the initialization of each vertex in the contracted graph is obviously parallel. The addition of edges between the vertices may be parallel assuming proper atomics or locks are used (such atomics do of course serialize the access to a specific component vertex). In a naive approach with an evenly distributed edge count between the components, the parallelism is bounded by the number of components in the system and the execution time of the componentization should be in the order of $O((|V| + |E|)/C)$, where C is the number of components.

Assuming a worst case cycle detection time of $O(|V|log|V|)$ as given as a sequential time by [14], we get the total execution time as $O(|V|log|V| + |E|/P + (|V| + |E|)/C + ClogC)$.

Note that for all practical applications, L is a per data structure constant, and the running time will be linear with the size of the data structure in terms of

the number of vertices and arcs.

With the shown ordering of the confined components in a strictly ordered orthogonal graph, it is important to note that if a data structure is mapped into a grid based on the confined components, it is possible to maintain the data dependencies of the original traversal order.

3.5 Special Grids

For the theory introduced in this chapter there are both theoretical and practical applications. Already discussed is the notion of strictly ordered orthogonality, which is an application of the confined components introduced in this chapter. Strictly ordered orthogonality has been defined with the intention of pointer traversals being embeddable in n-dimensional arrays, where the number of edge disjoint vertex covering sets of confined components represents the dimensions in the array.

In terms of shapes, there are two grid types that we are interested in discussing: these are square grids and triangle grids. In both cases, these grids can be directly implemented with pointers, yielding dense mesh implementation. However, in many applications, computations are not defined on the direct implementation of these grids, but rather in an indirect way and on their adjacency matrix representations. These are especially common in high performance code libraries. For instance, in finite element applications the elements are typically loaded as a triangular or tetrahedral direct grid structure and after assembling the stiffness matrix, the computations switch to using an adjacency matrix. Note that in general, adjacency matrices are basically sparse rectangular grids.

3.5.1 Complete Rectangular and Triangular Grids

Strictly ordered orthogonality of complete square and triangular grids can easily be computed in polynomial time, as shown in Theorems 14 and 26.

An important property of square grids is that there is a two dimensional embedding of such grids so that pointer traversals can be translated into counted loops, with one index per dimension. However, for complete triangular grids (see Figure 3.3), the situation is a bit more complicated. This comes from the introduction of row alignment issues. For example, if the triangular grid is defined as given in the gray graph in Figure 3.3, one of the orthogonal arc sets is the set of diagonal arcs, pointing backwards in next row, this introduces back-dependencies that we want to avoid. This can either be avoided by con-

structing the grid in a way where this will not happen (i.e. as the white graph in the same figure), or by skewing the grid, so that the diagonals become the down arcs. Note that this is not a direct embedding and empty elements need to be inserted on the edges after the skewing to map this directly.

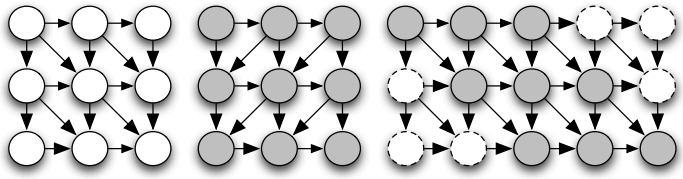

Figure 3.3: Triangular Grids. The fully white grid has the right and down directions orthogonal, the gray grid has its right and diagonal directions orthogonal. In the right hand graph, the middle graph has been skewed and empty elements inserted to ensure that the orthogonal directions line up and can be traversed by modifying only one iteration index.

Triangular grids have a three dimensional counterpart called a tetrahedral grid. Tetrahedral grids can also be defined in a way that allows for the determination of three orthogonal arc sets, for a simple example with twelve tetrahedrons, see Figure 3.4. Iterations over these arcs can then also be mapped into indexed loops where three indices are used. Essentially, a right angled tetrahedral grid can be mapped into a set of cubes (forming an overlaying cube grid), the dimensions of the cube represent the orthogonal arc sets which can be mapped to a single index increment and traversals in the remaining arcs can be replaced by a incrementing either two or three dimensions. Note that skewing needs to be handled in an even more intricate way than was the case for the triangular grids, and may not always be possible.

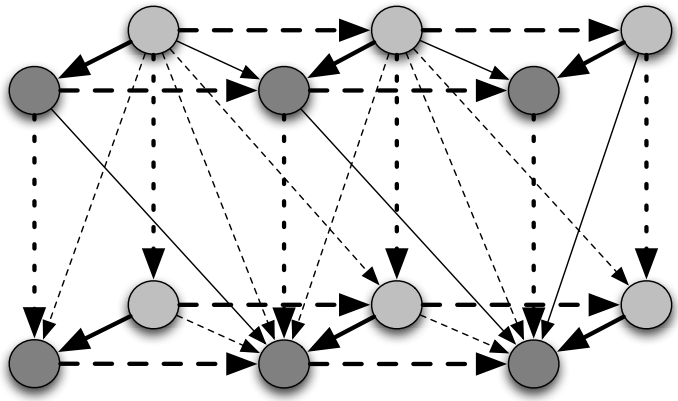

Figure 3.4: Orthogonal Tetrahedral Grid Building Element. The edge styles represent the different dimensions in the grid. In this case, the fat solid lines, the fat long striped lines and the fat short striped lines indicate the orthogonal dimensions in this grid (i.e. depth, horizontal and vertical dimensions).

3.5.2 Sparse Square Grids

Most sparse square grids arise from adjacency matrices in the code. These grids are strictly ordered orthogonal as defined in this chapter, and the underlying grid may be easily decomposable (in orthogonal arc sets). There are two options for decomposing these grids, one is to attempt decomposing the sparse grid itself, the second option is to map the sparse grid into the direct implementation and to decompose the direct grid[5].

Deriving the mapping to the direct grid is not straight forward and is rather artificial with respect to the algorithms working on the grids (row wise traversals become iterations on vertex neighbors, and column traversals become iterations over the vertices), i.e. a traversal over a sparse grid's rows and columns becomes a traversal over the direct grid's rows and columns and all neighbors of the visited nodes. Note that, in order to derive a mapping to the original complete grid we would need to find the rows and the columns in the adjacency matrix. This means that the decomposition of the direct grid in

[5]A direct implementation is a direct representation of a matrix in matrix-form; this should be contrasted to indirect solvers that map a matrix into an adjacency structure

strictly ordered orthogonal arc sets is dependent on the decomposition of the sparse adjacency matrix grid.

Presumably, for analysis purposes, we would want to insert empty fill elements in the sparse square grid in order to be able to embed it into a two dimensional grid. This is similar to how the triangular grid with back pointing dependencies was handled (bottom graph in Figure 3.3), except that the empty fill elements would also be inserted between the elements (not only outside the main grid). This insertion strategy serves as a way of applying optimizations designed for codes working on direct grids to codes working on indirect grids.

3.5.3 Exploiting Knowledge on Pointer Types

In many cases recursive pointer types are used to implement graph structures. This is important since the programs often constrain the purpose of each pointer field to point in a specific dimension. We can use this property to infer a decomposition into potentially orthogonal arc sets. The orthogonality assumption can in turn be verified with an efficient algorithm (see Theorem 26). For example, in a sparse matrix-vector multiply code, a common implementation is to use a record containing two pointer fields (right and down). This record may in turn be used to construct a graph as the one illustrated in Figure 3.5. Note that without any kind of verification of the assumed shape, it would not be correct to assume that the structure represents a matrix, since any pointer may alias the other pointers in the structure or may be part of some other structure (e.g. a tree or some generic DAG). In such cases, embedding in a two dimensional grid would not be a good choice.

We can also use the property of directed arc chains to eliminate pointer fields from the analysis and support anti-parallel feedback arc sets when such sets exist in a pointer structured graph. Despite the fact that such arcs introduce cycles (for example the set of all up and left pointers form an anti-parallel feedback arc set in a bidirectionally linked grid, that is, *left* is anti-parallel to *right*, and *up* is anti parallel to *down*, and removing these pointers will leave the remaining graph as a DAG), verifying that a named pointer field is anti-parallel with another pointer field is obviously linear with the number of vertices in the graph.

3.6 Potential Applications

Confined components and the orthogonality of arc-sets have several compiler applications that could almost be directly implemented in many existing com-

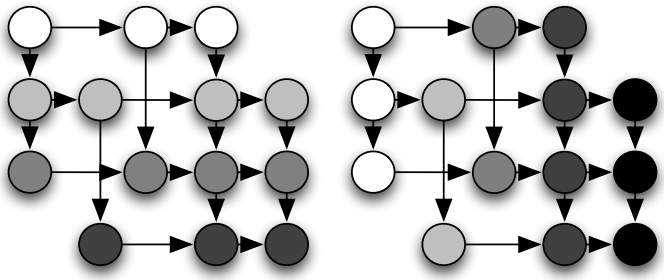

Figure 3.5: Strictly Ordered Orthogonal Decomposition of a Sparse Square Grid. Each confined component in the decomposition is indicated using a separate shade of gray.

piler tool chains. The fundamental area where the concepts can be applied is the optimization of pointer structured code and data structures. This type of optimization, where for example objects are being moved around or pointers are added to or removed from a structure, is called restructuring.

It should be noted that in general it is not possible to restructure data during compile-time since data in most cases is read from files that the user may modify. Because of this, restructuring of data needs to be done during run-time; but the compiler may help by analyzing the program and data structures in order to assist with the generation of restructuring code, or by restructuring the data structure types themselves (e.g. eliminating pointers in a structure or replacing them with indices).

There are three key problems that need to be solved in order to restructure pointer-based data structures. These are data structure memory *layout*, chained pointer *dependencies* and the pointer *aliasing* problems.

In the first case, when a pointer based structure is created (often involving a recursive data structure), the order in which elements are read from file will in many cases impact the memory order of the elements. An example that illustrates this is a common pattern from sparse matrices, where a structure is used for each element in the matrix. This structure is then usually linked to the next in row and the next in column elements. Depending on how the system's standard memory allocator works (e.g. malloc), the elements will be created one by one in an increasing memory order. If the order of the elements are the same as the logical structure, that is element $(0, 0)$ first, then $(0, 1)$

and finally (m, n), the matrix elements will be allocated in a row-wise order since the file the matrix is read from is ordered in that way. If the elements in the file are unordered the right element pointer will not necessarily point to the next physical element.

For the problem relating to chained pointer dependencies, it should be obvious that in order to access an arbitrary element within a pointer linked structure, a linear traversal of the pointer chain is needed. Such dependency chains directly impact parallelization and vectorization opportunities as a pointer chain cannot easily be split up through a divide and conquer like pattern.

The aliasing problem is dealing with the question of whether two pointers can point to the same object. Some languages such as Fortran make this implicit, others like C allow the programmer to have a more relaxed notion and to tell the compiler when there will be no aliasing. In many cases this is not practical, so several inter-procedural algorithms for doing points-to analysis have been devised such as Bjarne Stensgaards analysis [47] in order to automate the aliasing resolution. Such algorithms typically classify pointer dependencies as *will not alias*, *may alias* and *will alias*. This allows the compiler to eliminate redundant pointers and assume that objects are independent in many cases.

These three problems (order, dependencies and aliasing) are all very interesting by themselves, and various solutions have been developed that each deal with one of the problems and not the others. As will be shown here, the confined components and the notion of orthogonal edge sets may help to deal with all three of these problems.

For the applications discussed in this section, we predefine a number of types, though we only show the pointer fields in the types as these are the relevant fields for the graph structure.

```
typedef struct cell_t {
  cell_t *right;
  cell_t *down;
} cell_t;

typedef struct {
  size_t rows;
  size_t cols;
  cell_t **row;
  cell_t **col;
} matrix_t;
```

When the types are used in the applications below, we assume that it is known that the rows and the columns are vertex covering confined component

sets. Implicitly from this, we are aware that the recursive pointer chains will end in NULL at some point (or the pointer graph would not be a DAG).

We must also stress, that control flow information needs to be considered by the compiler before doing any transformations like the ones mentioned. For example, the component traversals may depend on additional variables and for example loop interchange may not be suitable without proper control flow information. Such control flow needs to be consider together with the orthogonality and confined component formalisms.

3.6.1 Linearization

For large pointer linked data structures, memory bandwidth may be a severely limiting factor. This is especially problematic when data structures visited in sequence are not adjacent in memory. The memory system will in many cases have to fetch unnecessary data outside the structure that is being loaded, that are not part of the next object. In addition to this, pointer based structures are not directly position independent (since they contain pointers). Both of the issues may be alleviated, by applying linearizing transformations (as for example described in [53]).

For a pointer linked data structure, a linearization pass would relocate the objects into an array, where the recursively typed pointers may be replaced with array indices. The transformation, while obviously being trivial to apply on a singly linked list, is non-trivial to apply on complicated pointer linked structures. For example, in the linked list case, we can simply traverse the list and relocate the objects, but for an object with more than one recursively typed pointer the object may represent any kind of graph structure, so the order of the linearization is not trivial, and it may not make sense in many graphs.

By showing a pointer linked structure to be orthogonal, we can apply the linearization step on two-dimensional grids. This is done by linearizing based on one of the dimensions (i.e. vertical or horizontal) in the grid. Figure 3.6. illustrate the transformations on the data structure.

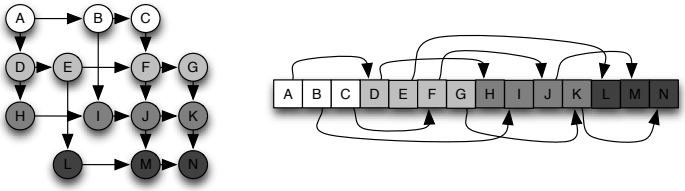

Figure 3.6: Linearization of an orthogonal grid. The left figure shows the data structure's logical layout (the nodes can be spread out on any kind of location), the right hand graph shows the structure in its post linearization physical layout.

Consider the following code iterating over the matrix type defined above:

```
for (int i = 0 ; i < m->row_count ; i ++) {
  cell_t *p = m->row[i];
  while (p) {
    p = p->right;
  }
}
```

In order to exploit the linear behavior of the right pointer member, we make two transformations on the code, the first step moves the inner loop into a pointer increment based iteration:

```
for (int i = 0 ; i < m->row_count ; i ++) {
  cell_t *p = m->row[i];
  while (WITHIN_ARRAY(p)) {
    p ++;
  }
}
```

The next step is to rewrite the while loop into a for-loop:

```
for (int i = 0 ; i < m->row_count ; i ++) {
  cell_t *p = m->row[i];
  for (int i = 0; i < ARRAY_LEN(p) ; i ++) {
    // Replace dereferences of p with p[i]
  }
}
```

In order to accomplish this, the language or the optimizing runtime would need to provide information on how to determine array membership and lengths. Although, the C-language does not provide this information, the optimization relies on the fact that confined components have already been detected. Therefore assume that this information is available. As the pointer linked structures are typically built dynamically (by for example loading a file), it would be necessary to take this into account. We suggest that the compiler should be able to determine when a structure is complete, either through analysis or by programmer intervention.

3.6.2 Advanced Pointer Elimination

While the linearization mentioned earlier provided a simple type of pointer elimination, it still does not eliminate pointer fields in the structures. If pointers are unnecessary they will take up memory bandwidth, therefore it is of interest to get rid of some of the pointers completely.

A related problem is the position dependencies of the data structure, such dependency does for example prevent the offloading of computations within an heterogeneous system (such as for example GPUs) that work with different address spaces. Consequently, it would be interesting to eliminate unused pointer fields in a structure in order to save memory bandwidth and to make the structures position independent.

If pointers are used to traverse a chain, they can be indexified by building on the linearization step mentioned previously. Consider a pass through the data structure that replace all the pointers with indexes within the linearized structure instead. This structure is immediately position independent and it is now relatively easy to move the pointer based structure to a GPU.

Such transformations require access to the linearized structure's base pointer, and this may not always be possible. We could solve this by cloning functions, adding extra arguments with the base pointers (much like the work on automatic pool allocation, where pool pointers where automatically added to the function's parameter lists).

In order to replace the pointers within the linearized region, we need to introduce new types.

```
typedef unsigned long cell_idx_t;

typedef struct cell_t {
  // Right eliminated
  cell_idx_t down;
} cell_t;

typedef struct {
  size_t len;
  cell_t elements[];
} cell_array_t;

typedef struct {
  cell_array_t *cells;
  cell_idx_t start;
  cell_idx_t end;
} cell_ptr_t;

typedef struct {
  size_t rows;
  size_t cols;
  cell_ptr_t *row;
  cell_ptr_t *col;
} matrix_t;
```

The cell array can be used by the linearization pass to store all the cell objects. This is used by the cell pointer type, which can be used when dealing with cell pointers on the stack, or in other types, the cell pointer type has a start index and an end index in order to signal the start and end of a confined component within the array (in this case, the components are either rows or columns).

Assuming that the cell type is responsible for the majority of the data traffic, the setup shown here will reduce the memory bandwidth requirements by eliminating one pointer field (we can potentially also reduce the size of the index type to 32 bit if the application can be limited to work with 4 billion connected cell objects at most). In addition to reducing the bandwidth requirements, the matrix type will be mostly position independent (the only exception being the cell array pointers, but those can be reduced in numbers

by for example factoring out the array pointer from the cell pointer type and storing it separately in the matrix type).

Assuming that the application only stores a single double value in the cell objects the transformation has eliminated almost a third of the memory bus traffic stemming from the matrix traversals.

The suggestions in this section do depend on the linearization optimization described in the previous section. As such, it requires development of language constructs or control flow analysis that can determine when a structure can be rewritten in the given form (for example, the compiler must be able to determine that no additional objects are inserted in the structure and that the *right* pointers are constant). In order to execute a pointer based program on a GPU, additional technology would be needed that could generate working GPU code for parts of the program, while GPU compilers naturally exist and are well used. Some work has been carried out in the area of allowing unmodified C-code to execute on a GPU such as for example [2] that describes a compiler capable of generating CUDA code from dense C-based loops (that can be analyzed using the polyhedral model). We are not aware of any compilers that can take a sparse pointer based C-code and generate GPU code for parts of the program. As such, we believe that linearization in combination with advanced pointer elimination techniques could be a powerful transformation opening up the possibility for GPU based computation on some pointer based codes.

3.6.3 Replacement Algorithms, Garbage Collection and Leak Detection

Replacement algorithms and policies have been extensively studied by others, one of the most well known is the Least Recently Used (LRU) replacement policy that will replace the oldest reference (in the cache or Translation Look aside Buffers (TLB) for example) whenever the cache unit is full. Although the LRU policy is difficult to implement in hardware, many of the cache replacement algorithms are derived from LRU such as for example the pseudo LRU policy and others [13, 18, 42]. However, LRU and its derivatives are not optimal and it is easy to write code that will suffer if LRU is used. The optimal algorithm for replacement policy was described by Belady[3] and can be described as replacing the unit that will be needed the furthest time in the future, this however requires that the future is well known and can only be used as is in some severely restricted cases.

Related to the concept of cache replacement algorithms is the notion of garbage collection and leak detection. These concepts are similar to cache

replacement in the aspect that they are also dealing with reachability, although for replacement there is not necessarily a strict reachability requirement but rather a temporal requirement that the replaced objects should not be reached soon.

In either case, by applying the confined component detection and matching the links to a *successor* component within the iteration space in the code, we should be able to apply an informed replacement policy, or run a potentially efficient garbage collection / leak detection algorithm. In the former case, components that are exited by some iteration will be evicted from the caches, and in the second case, a component that is exited during an iteration could for example have a component-reference count reduced. Note that the common reference counting issues where cycles need to be specially handled does not apply in this case as components by definition are not part of cycles when contracted.

Consider the following line of code:

```
p = p->down;
```

In this case we assume that *down* always leave a component, though it may be possible to have more complex exit criteria we have at this moment not explored the area and as such this is left as future work.

Consider the act of leaving a component, it is likely that the previous component is no longer needed for some time, in this case the entire component can be evicted from cache. If the hypothesis that the down dereference leaves a confined component is correct, it may be suitable to transform the line into the following:

```
p_tmp = p;
p = p->down;
flush_component(componentof(p_tmp));
```

Naturally, whether or not this transformation makes sense would depend on the component, cache and TLB sizes. In this case, when a component is flushed, one stack root pointing into some compoenent is also replaced. If the code leaves a component, it may be necessary to consider what this entails for garbage collection and leak detection (the latter being similar but more of a debugging tool for gc-free systems).

In these cases it may be interesting to consider whether reachability information should be computed on component graphs instead of individual

objects. Essentially, this could potentially allow higher level GC and leak detection systems. The performance of these systems may be improved as they would no longer need to scan through every single heap object. Instead they can walk the contracted graph of components, which hopefully contains fewer vertices than the full graph.

The following example illustrates how some object pointer is declared on the program stack and then used in a reference counting system[6]. The difficult part is obviously not in the actual reference counting, but rather in the detection of the fact that p is pointing into a component of some sort.

```
{
  node_t *p = ...;
  inc_ref(componentof(p));
  ...
  dec_ref(componentof(p));
}
```

In order for this to work properly a runtime function for determining the component of an object would be needed. None of these applications rely on linearized components as described in section 3.6.1, although it should be noted that linearization would probably simplify the runtime.

3.6.4 For-each Detection and Loop Interchange

Given that the confined component sets may specify dimensions of a graph, it may be possible to determine that loops are actually for-each loops. Especially in the case of orthogonal arc sets, it may be possible to permute some loops that iterate over the main dimensions of the orthogonal structures. For example, if the rows are linearized and a loop is found to be traversing the columns, it would be better if the loops where interchanged and the iteration done over the rows instead. The exact safety of these transformation would naturally depend on the control flow of the application, and whether or not all row and column headers can be derived.

Consider the following loop, which is representative to some sparse matrix codes:

[6]Note, that we don't have to bother about the issues reference counting have with cycles since by definition, the confined components are cycle free.

```
for (int i = 0 ; i < m->row_count ; i ++) {
  cell_t *p = m->row[i];
  while (p) {
    p = p->right;
  }
}
```

Consequently, if the compiler would hypothesize that row[] contains all roots in a vertex covering set of confined components formed by following the right pointers, and similarly that col[] contains all the roots forming a vertex covering set set of confined components with the down pointers. By determining that the two sets are orthogonal, it is possible to do loop interchange on the the loop. Then the loop can be transformed into:

```
for (int i = 0 ; i < m->col_count ; i ++) {
  cell_t *p = m->col[i];
  while (p) {
    p = p->down;
  }
}
```

This assumes that there is a way to bind the *col count* and *row count* variables to the actual array lengths. This is implicit in many programming languages (but not in C).

Like normal loop interchange, this optimization primarily targets the memory order of the dereferenced elements in the matrix. For sparse matrices, the memory order is typically dependent on the input to the program, so a good solution in this case would be to emit multi-versioned code (one for row major and one for column major orderings).

In addition to this, parallelism in the outer loop could potentially be exploited as it is clear that the inner loops are independent of each other. The code would then be trivially transformed into the following sequence:

```
parallel_for (int i = 0 ; i < m->row_count ; i ++) {
  cell_t *p = m->row[i];
  while (p) {
    p = p->right;
  }
}
```

3.6.5 Implementation Issues

As it is computationally expensive to detect confined components, we cannot expect that an implementation would try to do this during compile- or runtime without having constraints on how the components may be built.

We could assume that for example orthogonal edges will be different pointers in a structure (e.g. right and down pointers in a sparse matrix). Although subsequently verifying this during runtime could potentially be expensive as there is no conceptual difference between say a sparse matrix node and a binary linked list node. Both have two pointers embedded in themselves and cannot be distinguished from each other. Two of the most obvious ways to overcome this is to let the programmer specify additional meta-information through attributes or pragmas in the source code, or through the less general way where the compiler simply analyzes the language of the source code. If it finds fields named right and down it makes one assumption, but if it finds fields named right and left it makes another.

3.7 Summary

In this chapter we introduced a new graph theoretical concept, the confined component and showed several applications that could be used in a compiler (in combination with a run-time system). Though the MINIMUM CONFINED COMPONENTS problem was shown to be NP-complete, this restriction does not apply to practical cases where we do not need to find the components, but we have only to verify that we have a given decomposition of the components.

An efficient algorithm able to find the vertex covering sets of confined unilateral components was introduced. This algorithm was proven to be correct and optimal on some important graph shapes.

Strictly Ordered Orhogonality was introduced and an algorithm was given that is able to verify an orthogonality hypothesis in polynomial time.

We believe that a compiler that would take this theory into account would open up new venues into, for example, automatic parallelization of sparse irregular pointer based codes.