Karlsruhe Institute of Technology

Comparing Deductive Program Verification of Graph Data-Structures

Bachelor's Thesis of

Jelle Kübler

at the Department of Informatics
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. Bernhard Beckert
Advisor:	Dr. rer. nat. Mattias Ulbrich
Second advisor:	Dipl.-Inform. Michael Kirsten

4 June 2018 – 3 October 2018

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Karlsruhe, 1 October 2018

. . . . . .

(Jelle Kübler)

Abstract

Graphs and graph-based algorithms are ubiquitous. With their omnipresence, it is important that these graph-algorithms work correctly. To ensure that those algorithms behave correctly, they can be formally verified. Deductive program verification is used, to show that a program fulfills a formal specification of its behavior for every valid input, which is a non-trivial task. Also, there is not only one single implementation of graphs. Graphs can be represented by a variety of data structures that all have their advantages and disadvantages in terms of e.g. runtime or memory-usage.

It is unclear, where potential bottlenecks might occur during the verification of graph-algorithms. Throughout these data structures, there are differences with respect to their suitability for deductive program-verification of graph-algorithms. Knowing with which graph-representations graph-algorithms are easier to verify, provides an additional decision aid, when selecting an implementation.

In this thesis, we look at one specific algorithm: depth-first search. DFS is a simple algorithm for traversing graphs. Many graph algorithms are based on DFS. So, despite its simplicity, the verification of depth-first search is of interest. We specify and verify two different implementations of depth-first search. To do so, we use the formal specification language JML, to specify Java programs, and the KeY tool for deductive verification. We study each implementation with four different graph-representations. The considered graph data structures are the adjacency matrix, adjacency array and adjacency lists as well as a linked data structure using objects and pointers. As implementation of depth-first search, we use the recursive versions, using recursive method-calls, and the non-recursive version, iterating a loop instead of recursive calls. We show that the recursive version is verified for all four data structures, while the non-recursive version is only verified for the adjacency matrix version.

Additionally to the verification of DFS with different data structures, we define the depth-first property of a graph traversal, in order to prove that the algorithm traverses the graph in a valid order.

Finally, we compare the verification-process of recursive and non-recursive depth-first search, for all four representations. We compare them in terms of different aspects, like the time effort of the verification process or the size of the resulting proof.

In the end, this thesis discusses advantages and disadvantages of the four graph-representations, during the verification of a graph-algorithm. Additionally, we see how the recursive implementation of DFS can be easier to verify with KeY, than the non-recursive version. Furthermore, we give a general definition of the depth-first property of a graph traversal and a set of rules to handle bounded sums in KeY.

Zusammenfassung

Graphen und Graph-basierte Algorithmen sind allgegenwärtig. Aufgrund dieser Allgegenwärtigkeit, is es wichtig, dass diese Algorithmen korrekt funktionieren. Um das zu gewährleisten, können die Graph-Algorithmen formal verifiziert werden. Deduktive Programm-Verifikation wird verwendet, um zu zeigen, dass ein Programm eine formale Spezifikation seines Verhaltens, für jede gültige Eingabe erfüllt, was keine triviale Aufgabe ist. Außerdem gibt es nicht nur eine Implementierung von Graphen. Diese können in einer Vielfalt von Datenstrukturen dargestellt werden, die alle ihre Vor- und Nachteile; z.B. bezüglich Laufzeit oder Speicherverbrauch haben.

Es ist unklar, wo potentielle Engstellen im Verifikationsprozess auftauchen werden. Unter diesen Datenstrukturen gibt es besser und schlechter geeignete, wenn es um deduktive Programmverifikation von Graph-Algorithmen geht. Mit dem Wissen darüber, welche Datenstrukturen sich besser eignen um einen Graph-Algorithmus zu verifizieren, bekommt man eine zusätzliche Entscheidungshilfe beim Auswählen einer Implementierung.

In dieser Arbeit untersuchen wir dies für einen konkreten Algorithmus: Tiefensuche. Tiefensuche ist ein simpler Algorithmus, um einen Graphen zu traversieren. Viele komplexere Graph-Algorithmen basieren auf Tiefensuche. Somit ist die Verifikation von Tiefensuche, trotz seiner Einfachheit, von Interesse. Wir verifizieren zwei verschiedene Implementierungen von Tiefensuche in Graphen. Dazu verwenden wir die formale Spezifikationssprache JML, um Java-Programme zu spezifizieren, und das KeY-System für die deduktive Verifikation. Beide Implementierungen werden wir für vier Graph-Repräsentationen untersuchen. Die verwendeten Graph-Datenstrukturen sind Adjazenz-Matrizen, -Felder und -Listen, sowie eine verzeigerte Datenstruktur. Als Implementierung von Tiefensuche, verwenden wir eine rekursive Version, die rekursive Methodenaufrufe verwendet, und eine nicht-rekursive Version, die statt rekursivem Aufruf einen Stack iteriert. Zusätzlich zu der Verifikation mit diesen Datenstrukturen, definieren wir die Tiefensuch-Eigenschaft, um zu zeigen, dass der Algorithmus den Graphen in einer gültigen Reihenfolge traversiert.

Schlussendlich vergleichen wir den Verifikationsprozess sowohl für rekursive, als auch für nicht rekursive Tiefensuche, jeweils mit den vier genannten Datenstrukturen. Verglichen werden sie unter verschiedenen Aspekten, wie der Dauer des Verifikationsprozesses oder der Größe des Beweises.

Damit diskutiert diese Arbeit die Vor- und Nachteile der vier Datenstrukturen, während der Verifikation eines Graph-Algorithmus. Außerdem zeigt sie, dass die rekursive Implemetierung mit KeY einfacher zu verifizieren war als die nicht-rekursive. Außerdem geben wird sowohl eine allgemeine Definition der Tiefensuch-Eigenschaft einer Graph-Traversierung gegeben, als auch ein Satz von Regeln, um in KeY mit Summen umzugehen.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1 Introduction

These days, graphs and graph-based algorithms are ubiquitous. Graphs appear as social network graphs, syntax trees of formal languages or represent traffic networks. They support efficient calculation of centrality measures, a programs name-spaces or shortest paths between vertices. Due to their omnipresence, it is important that these graph-algorithms work correctly. Therefore, these algorithms can be formally verified, so they can be proven to fulfill certain properties or standards. Also, there is not only one single implementation of graphs. Graphs can be represented by a variety of data structures, some of which may be more efficient in terms of runtime or memory-usage, while others may be more straightforward to implement.

Depth first-search (DFS) is a simple algorithm for traversing graphs whilst marking visited nodes. Many graph algorithms, e.g., Dijkstra‘s algorithm for finding shortest paths or Tarjans algorithm for computing strongly connected components, are based on DFS. So, despite its simplicity, DFS his of interest.

The goal of this thesis, is to compare the deductive program-verification of graph-algorithms, using different graph data structures. Deductive program-verification means full functional, formal verification of a program, based on a formal specification.

To approach this goal, we will present the verification-process of one graph-algorithm on a set of four chosen data structures. We will use JML (Java Modeling Language) [LPC+08], a specification language for Java, to specify the implementations and the program verification system KeY [ABB+16] to verify them. As algorithm, we will look at two implementations of depth-first search (i.e. depth-first traversal (DFT)), as it is a simple, yet important graph-algorithm. The graph-representations that we will consider are adjacency matrix, adjacency array and adjacency lists as well as a linked data structure using objects and pointers. Additionally, we give a formal definition of the depth-first property of a graph traversal, in order to specify, when an algorithm traverses a Graph in a valid DFT order.
We will compare the suitability of DFT with these four graph-representations in the verification-process and argue, where certain advantages and disadvantages of these representations stem from. To estimate their suitability, we will compare the data structures under different aspects like the time effort of the verification process or size of the resulting proof.

Chapter 2 gives a short summary of graph theory in section 2.1 and introduces the four data structures that we will consider in section 2.2. Also the two implementations of depth-first traversal, recursive and non-recursive DFT are presented in section 2.3. Additionally it contains an introduction to JML and KeY in sections 2.4 and 2.5.
Chapters 3 and 4 present the general implementation, specification and verification of recursive and non-recursive DFT, as well as the specific verification-process for each graph-

representation. In chapter 5 we will discuss, how the depth-first property of a graph traversal can be specified and verified in detail.

Finally, we will evaluate the time effort and proof statistics for each verification-process and compare the advantages and disadvantages of the four data structures in chapter 6. Chapter 7 contains the conclusion where we will look at the results and give an outlook on potential future work.

2 Theoretical fundamentals, Tools and Techniques

2.1 Graph Theory

Graphs describe structures, consisting of a set of points V (called vertices) and a set of lines E between those points (called edges). With a graph, objects can be modeled as vertices and associations between objects can be modelled as edges. For example, traffic-networks can be represented by a graph, by modeling roads as edges and crossroads as vertices.

In an undirected graph $G = (V, E)$, E contains undirected edges $e = \{u, v\}$, connecting two vertices u and v. For an edge $e = \{u, v\}$, there is no order of the vertices, which is why it is called an undirected edge.

Figure 2.1 shows two undirected graphs, each consisting of four vertices: a,b,c and d. Although both contain the same vertices, the two graphs are not the same, since they contain different edges.

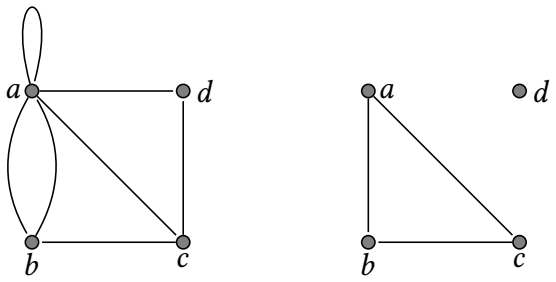


Figure 2.1: Two undirected graphs, both consisting of four vertices: a,b,c and d

The right graph in figure 2.1 can be created, if we take the left graph and remove one edge (a, b), the edges (a, d) and (c, d) as well as the loop (a, a).

Definition 1 (Subgraph) *A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$, if it can be created by removing edges from E or vertices and all connected edges from G:*

$$G' \subseteq G \Leftrightarrow V' \subseteq V \wedge E' \subseteq \{(u, v) \in E | u \in V' \wedge v \in V'\}$$

Definition 2 (Degree) *The number of edges connecting a vertex $v \in V$ with other vertices $u \in V$ is called the vertex's degree $\delta(v)$:*

$$\delta(v) = |\{\{u, w\} \in E | v = u \vee v = w\}|$$

In undirected graphs, there is no limitation to the number of edges between two vertices. Furthermore, edges with the same start- and end-vertex, called loops, or isolated vertices with no edges, are possible.

Definition 3 (Simple graph) *A graph $G = (V, E)$ is called a simple graph iff E contains neither multiple edges between two vertices nor loops.*

In figure 2.1 the left graph contains a loop at vertex a and multiple edges between a and b, hence it is not simple. The right graph, instead is simple. Its vertex d is called isolated, since it has no connected edge.

If its edges have a direction, a graph is called a directed graph.

Definition 4 (Directed graph) *A directed graph $G = (V, E)$ consists of a set of vertices V and a set of edges $E \subseteq V \times V$, where the edges $e \in E$ are pairs $e = (u, v)$. The pair provides an order of the vertices, implying the edge's direction: an edge $(u, v) \in E$ starts at vertex u and ends at vertex v.*

Undirected graphs can be modeled as directed graphs, by replacing every undirected edge $\{u, v\}$ with two directed edges (u, v) and (v, u) in either direction, as figure 2.2 shows.

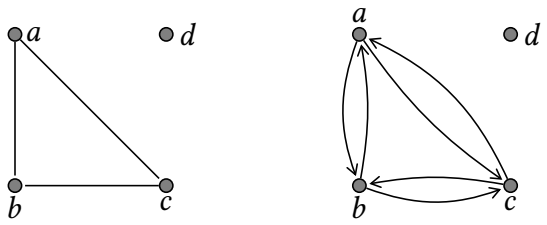


Figure 2.2: Undirected graph (left) modeled as directed graph (right)

In directed graphs, the degree of a vertex v can be divided into in- ($\delta^-(v)$) and out-degree ($\delta^+(v)$), the number of incoming and the number of outgoing edges.

Definition 5 (In- and out-degree) *For a vertex v in a directed graph $G = (V, E)$, the in-degree $\delta^-(v)$ is the number of incoming edges , ending at v:*

$$\delta^-(v) = |\{u \in V | (u, v) \in E\}|$$

The out-degree $\delta^+(v)$ of v denotes the number of edges that start at v:

$$\delta^+(v) = |\{u \in V | (v, u) \in E\}|$$

When using graphs, we often want to know, whether one vertex v is reachable from another vertex u, by repeatedly traversing edges (initially starting at u), until the destination v is reached. If v can be reached, then u and v are connected through a path.

Definition 6 (Path) *A path in a graph $G = (V, E)$ is a sequence $p = (v_0, \ldots, v_m)$ of vertices, so that two consecutive vertices v_i and v_{i+1} on the path are connected by an edge (i_1, i_2). The path $p = (v_0, \ldots, v_m)$ starts at vertex v_0 and ends at vertex v_m.*

Additionally to general graphs, there is a special type of graphs, called trees. Trees are graphs, where all vertices are reachable from one root, vertex. Also a tree does not contain cycles, so they can describe hierarchical structures.

Definition 7 (Tree, root, forest) *A graph $G = (V, E)$ is a tree iff there is exactly one vertex $r \in V$, called the root, with no incoming edges $\delta^-(v) = 0$. Also, every other vertex $v \in V, v \neq r$, has exactly one incoming edge: $\delta^-(v) = 1$. Additionally, there must be a path from the root r to every other vertex $v \in V$.*

A graph is a forest, if it consists of multiple, unconnected trees as subgraphs.

Definition 8 (Ancestor, descendant, parent, child) *Let u and v be vertices in a tree $T = (V, E)$ that are connected through a path $p = (u, \ldots, v)$. Then u is called an ancestor of v and v is called a descendant of u.*

If p is an edge from u to v: $p = (u, v) \in E$, then u is also called parent of v and v is called a child of u.

Figure 2.3 shows an example tree with root r. It contains an exemplary path from r to w, which is colored blue. The vertex t is a descendant of the vertex s and the root r is ancestor of all other vertices. In the example, the vertex u is the parent of the vertices v and w, which are the children of u.

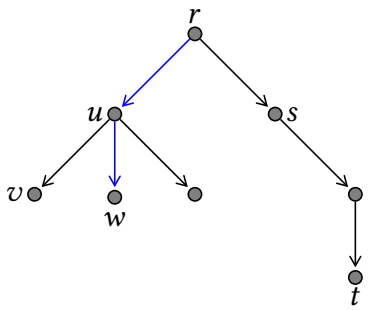


Figure 2.3: Example of a tree with root r

2.2 Graph Representations

If we want to use a graph in a program, there are several data structures that can be used to represent it. The two most popular ones are adjacency lists and adjacency matrices. Moreover, we will consider a compressed versions of an adjacency matrix (the adjacency array) and a linked data structure, to represent a graph. Each of them has its own advantages and disadvantages, which are mentioned in sections 2.2.1 to 2.2.4.

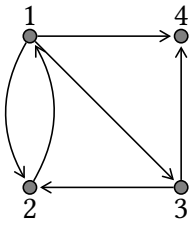


Figure 2.4: Example of a simple directed graph

The simple, directed graph in figure 2.4 will serve as example graph in the following subsections. Each of those sections describes a different graph data structure and provides an illustration of the example-graph in the corresponding graph-representation.

2.2.1 Adjacency Lists

Adjacency lists consist of an array a_V, whose entries each point to a list. The indices of a_V represent the graphs vertices $v \in V$ and the list $a_V[v]$ holds adjacent vertices to v. With an adjacency list, it is possible to add edges to the graph in constant time. By accessing the list $a_V[v]$, v's neighbors can also be retrieved in constant time. [MS08]

Figure 2.5 shows the adjacency list representation of the graph in figure 2.4. For example, the edges (3, 2) and (3, 4) are represented by the third list ([2, 4]) which contains the destinations of those edges that start at 3. Since there are no edges leaving vertex 4, its adjacency list is empty.

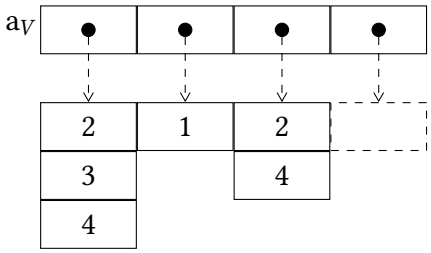


Figure 2.5: Adjacency lists representation of graph 2.4

2.2.2 Adjacency Matrix

An adjacency matrix is a $|V| \times |V|$ matrix. An entry $a_{ij} \neq 0$ indicates that vertex i and vertex j are connected by an edge $e = (i, j)$. Hence the adjacency of two given vertices can be checked

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Figure 2.6: Adjacency matrix representation of graph 2.4

in constant time. Also, adding or removing edges in the graph can be done in constant time, by setting an entry a_{ij} to 1 or 0. Matrices can be efficient in terms of memory usage, if the graph is very dense, which means, the graphs number of edges $|E|$ is near $|V|^2$. In the opposite case of a sparse graph, a lot of entries are zero: $a_{ij} = 0$. In that case the other three data structures are more memory-efficient with a memory usage of $O(|V| + |E|)$. [MS08]

In figure 2.6 the 1 at row three and column two represents the edge (3, 2). The entries on the matrix's primary diagonal are 0, because there are no self loops in the graph.

2.2.3 Adjacency Array

To avoid storing all the 0-entries in sparse adjacency matrices, the data can be compressed into the adjacency array representation. Here, all the vertices' adjacency lists are concatenated and stored into one single array a_{adj}. Therefore, we must be able to determine the start and end of each vertex's adjacency list in a_{adj}. That is the purpose of the a_V-array: $a_V[v] = i$ and $a_V[v + 1] = j$ denotes that v's outgoing edges end in the vertices $a_{adj}[t]$ where $t \in [i, j)$. The exclusive upper index j of v's adjacent vertices is the lower index of the adjacency list of the vertex $v + 1$. If a vertex w does not have any adjacent vertices, then $a_V[w] = k$ equals $a_V[w + 1]$ and the range $[k, k)$ is empty. Since every vertex needs a lower and an upper index, there is one dummy entry at the end of a_V, which holds the upper index of the last vertex. Still, the dummy entry's index itself does not represent a vertex. Since all the edges' destinations are stored in one array, adding or removing an edge or vertex always leads to recalculation of both arrays a_V and a_{adj}. However, as only two one dimensional arrays are used, the required memory is reduced to $O(|\mathtt{V}| + |\mathtt{E}|)$. [MS08]

In the example figure 2.7, the neighbors of 3 can be found, by reading those entries $a_{adj}[i]$, where i is in $[a_V[3], a_V[3 + 1])$. That results to the entries in $[5, 7)$, which hold the vertices 2 and 4.

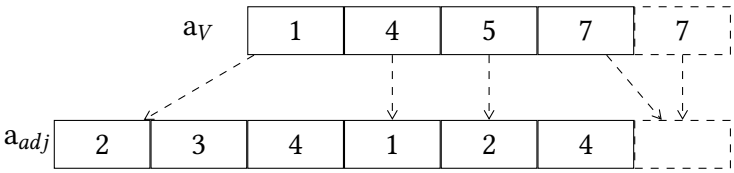


Figure 2.7: Adjacency array representation of graph 2.4

2.2.4 Linked Graph Data Structure

The linked graph data structure models vertices as objects. Each vertex-object has a list of pointers, which all point to adjacent vertices. Additionally, there is a list that keeps track of the graphs vertices. Similar to the adjacency lists data structure, it is possible to add or remove edges or get a vertexes neighbors in constant time. This graph-representation has a memory usage of $O(|V| + |E|)$. As this representation is an object-oriented approach, it is easier to read and more intuitive for object-oriented users. In figure 2.8, the vertex-object 3 holds pointers to the objects 2 and 4, which represent the edges (3, 2) and (3, 4). As there are no edges leaving 4, that vertex-object does not hold any pointers.

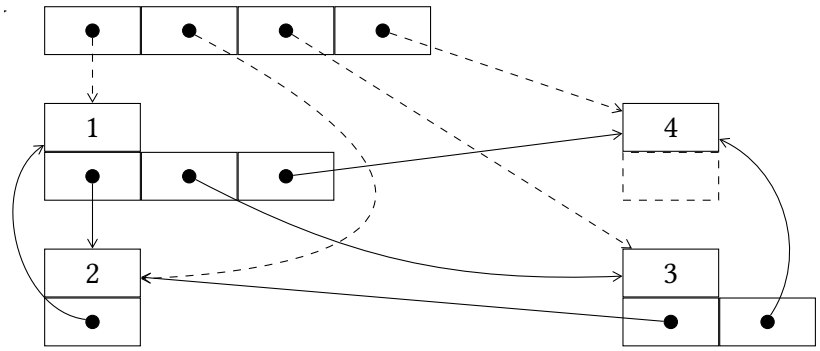


Figure 2.8: Linked graph representation of graph 2.4

2.3 Depth-First Search and Depth-First Traversal

Depth-first search (DFS) is a graph traversal algorithm. A graph traversal algorithm describes the process of visiting a graph's vertices. There are different types of graph traversal algorithms, which differ in the order, in which the vertices are visited. Depth-first search follows the strategy of traversing those edges that are leaving the last discovered vertex. When a yet undiscovered vertex is found, it is marked as visited, before exploring deeper into the graph. In case of traversing an edge that ends in an already visited vertex, the algorithm backtracks, until a vertex with not yet explored edges is reached and starts exploring one of them. In this way, the next visited vertex is always a child of the most recently visited vertex that still has unvisited children. The algorithm terminates, when all vertices that are reachable from the start-vertex, are marked as visited. [MS08]

In this thesis, we will work with depth-first traversal (DFT). DFT traverses all the graphs vertices, instead of only those that are reachable from a given vertex. To cover the whole graph, it is possible to start DFS multiple times, each time at another vertex that is not visited after the previous DFS.

DFS can be implemented in both a recursive (algorithm 1) and a non-recursive form (algorithm 2). The recursive form marks a newly reached vertex v as visited and then iterates the adjacent vertices of v, starting the *dfs*-routine at each of them. If $dfs(u)$ is called for a vertex u that has already been visited, the routine terminates without doing anything and backtracks to u's parent-vertex. The non-recursive form uses an explicit stack, on which it stores vertices. The non-recursive *dfs*-routine continues as long as there are vertices on the stack. In each iteration, it takes the vertex on top of the stack, marks it as visited (if it is not yet) and pushes all unvisited neighbors of that vertex onto the stack. In this way, the most recently pushed vertex is visited next. [Lar15, CLR+09]

In both cases, these routines can be started multiple times at vertices that were not reachable yet and are thus not visited. This ensures that the whole graph is covered, even if it is not strongly connected. In this way, depth-first traversal is performed.

When traversing a graph with DFS, the traversed edges can be divided into four classes: Edges, used to reach unvisited vertices, are called tree-edges. After DFS has finished, the subgraph G', containing only tree-edges, is a spanning tree of the graph. When performing DFT, G' is not a spanning tree but a forest of DFS-trees.

Non-tree-edges, which point from a vertex to one of its descendants in that tree are called forward-edges. Back-edges are those edges that point from a vertex to one of its ancestors. Forward- and back-edges always connect two vertices on a path in the DFS-tree. The rest of the graphs edges are called cross-edges. Neither of a cross-edges incident vertices is ancestor of the other, so there is no path in the DFS-tree containing both vertices. Figure 2.9 shows the four edge-types and their notation. [MS08]

Figure 2.10 shows the execution of depth-first search, in the example-graph from figure 2.4, starting at vertex a. It shows the state of the execution divided into 11 steps, where (1) shows the outset and (12) the final state. Visited vertices are shown in yellow and the vertex, from where DFS is currently searching, is marked red. Unexplored edges are shown in blue and the

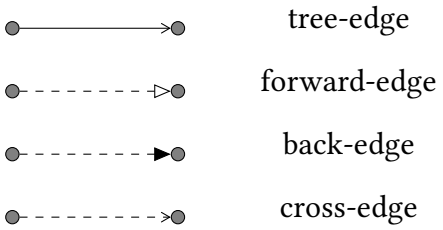


Figure 2.9: Notation of edge-types

edge, that will be traversed in the next step, is painted green. After an edge has been traversed for the first time, it is displayed in its matching classification (see figure 2.9).

In this way, every selected (green) edge, leading to a yet unvisited vertex, will become a tree-edge. In the example that are (a, b) in step $(2 \rightarrow 3)$, (a, c) in step $(5 \rightarrow 6)$ and (c, d) in step $(7 \rightarrow 8)$. When (b, a) is selected in (3), a has already been visited and is ancestor of b. Thus (b, a) is a back-edge. Also, when (c, b) is selected in (6) or (a, d) in (10), the destination vertex (b resp. d) has already been visited. Since neither c nor b is ancestor of the other, (c, b) is a cross-edge. And as a is ancestor of d, (a, d) becomes a forward-edge. In some states, there are no edges selected, which means that there are no undiscovered outgoing edges left, and the algorithem will backtrack in the following step along the incoming tree-edge.

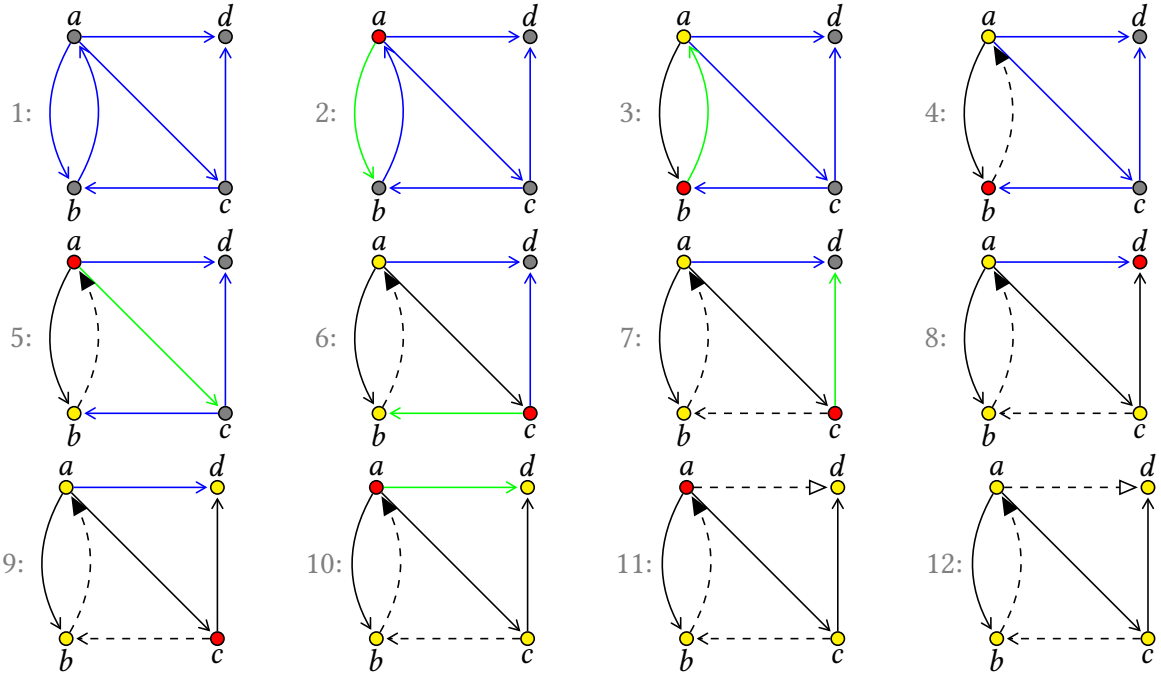


Figure 2.10: Exemplary DFS steps starting at a

In order to verify depth-first search, we will only consider simple directed graphs, since undirected graphs can be modeled as directed graphs (see section 2.1). The restriction on simple graphs is possible, since removing duplicate edges and loops does not change the graphs reachability properties:

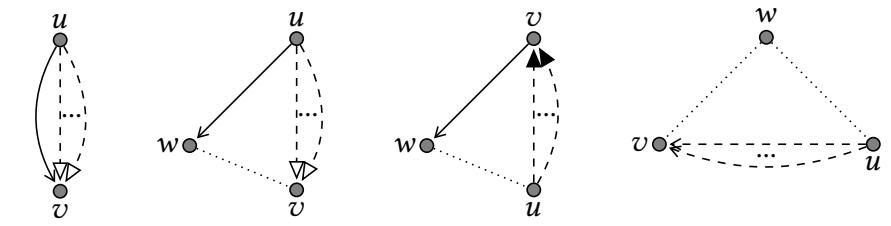


Figure 2.11: Cases of edge classification for multiple edges `(u,v)`

If a graph contains multiple edges $e_1 = e_2 = (u, v) \in E$, one of them can be a tree-edge and the rest are forward-edges, or all of them are forward-, back-, or cross-edges. Figure 2.11 shows the four possible cases of edge classification, assuming multiple edges from $u \in V$ to $v \in V$. The four cases are divided into cases *1.1* and *1.2* where u is visited before v and cases *2.1* and *2.2* where v is visited before u.

In case 1.1 of figure 2.11 v is visited directly after u, so one of the edges (u, v) is a tree-edge, the others are forward-edges, since v is a descendant of u. If there is at least one other vertex $w \in V$ visited between u and v, none of the edges (u, v) is a tree edge. Since v is still a descendant of u, all the (u, v)-edges are forward-edges, as case *1.2* shows. In the contrary case *2.1*, when u is a descendant of v, all the (u, v)-edges are back-edges. The last case *2.2* occurs, if neither u nor v is ancestor of the other vertex. Here the (u, v)-edges are cross-edges. The case that u is visited before v and (u, v) is a cross-edge can never occur, since DFS would visit all reachable vertices of u (including v) before backtracking to an ancestor of u.

In each case, removing duplicate edges does not change the DFS-tree (resp. DFT-forest). Also loops $e = (v, v)$ can never be tree-edges. Removing them together with duplicate edges, results in a simple graph, which still has the same reachability structure and therefore also the same DFS-tree.

Furthermore, timestamps can be used to indicate, when a vertex `v` was visited for the first time (pre-order index: `pre(v)`) and when it was left via backtrack (post-order index: `post(v)`). Figure 2.12 shows the example graph from figure 2.4 after the DFS traversal and the pre- and post-order-index of each vertex. Since a is the starting point, it has pre-order-index 1. After that, b is visited and receives pre-order-index 2. Since b has no unvisited adjacent vertices, the algorithm backtracks to a and b has now the post-order-index 3. Continuing this procedure will eventually provide a pre- and post-order-index for each vertex.

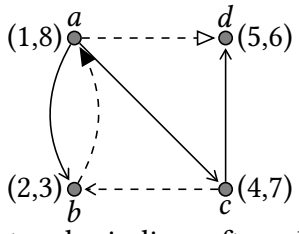


Figure 2.12: Pre- and post-order indices after a DFS traversal: (pre,post)

2.4 Java Modeling Language

The Java Modeling Language (JML) is a formal specification language that can be used to specify Java programs. The specification can be written as Java-comments and is based on the Java-syntax. JML also provides some new keywords like the universal (`\forall`) and the existential quantifier (`\exists`), as well as a notation for implication (`==>`) and equivalence (`<==>`), to extend the Java-syntax. With JML, the design by contract concept can be applied to Java programs. A software component's interface can be specified in JML, which forms a contract. Such a contract contains preconditions, which the user of the component has to meet, and postconditions, which the software component itself has to assure. Thus JML supports modular specification and verification.

To specify properties of a Java class, we can describe the functionality of its methods. Furthermore class-invariants can be used to describe properties of the class's fields.

Class-invariants are constraints to the class's fields, which define its legal states. All of the class-invariants must hold after the class's constructor finished. Also every method (except helper- and private methods) must preserve the invariants of its class. This means that before and after the execution of such a method, the class's invariants must hold. The example in listing 2.1 demands that `G` has at least one entry, that `cnt` is a value in [`0`,`G.length`] and that the `ord`-array has the same length as `G`. Those three conditions must be assured by the class constructor. [LPC+08]

```
public class GraphDFT_recursive {
  /*@ public invariant G.length > 0; @*/
  private /*@ spec_public @*/ boolean[][] G;

  /*@ public invariant 0 <= cnt && cnt <= G.length; @*/
  private /*@ spec_public @*/ int cnt;

  /*@ public invariant ord.length == G.length; @*/
  private /*@ spec_public @*/ int[] ord;

  /*...*/
```

Listing 2.1: Example of class-invariants

To simplify the specification, it is possible to define model-methods, as well as model- and ghost-fields. Model- and ghost-fields are both imaginary fields that serve only the purpose of formal verification. Model-fields are used to abstract from existing fields and methods, A model-field's value derives from the values of those fields and methods it abstracts. Ghost-fields instead, do not represent a specific field and can be assigned any value of its type, by using `set`-statements. Model-methods are imaginary methods, used in the specification, but they are not abstractions of other methods. [LPC+08]

Listing 2.2 shows an example of a model method `unvisited()`, which returns a location-set (a set of locations on the heap). The location-set is calculated as union of all locations in the `ord`-array that have a value of `-1`.

```
/*@ model \locset unvisited() {
              return \infinite_union(int i;
                   (0 <= i && i < ord.length && ord[i] == -1);
                   \singleton(ord[i]));
     }
  @*/
```

Listing 2.2: Example of a model-method

A method-contract consist of pre- (`ensures`) and postconditions (`requires`). A method is considered to fulfill its contract, if the postconditions are true after the execution of the methods body, assuming the preconditions were true upon method-call. When describing the postcondition, it might be necessary to compare the programs state before and after the execution. Therefore, the `\old` keyword provides access to values of fields before the execution. Also the `\result` keyword can be used, to argue about the result, returned by the method. If a method manipulates some of the class's fields, those can be listed in an `assignable` clause. Such an `assignable`-clause states that during the execution of the method all of the listed fields may change, while all other fields remain untouched. [LPC+08]

The example in listing 2.3 shows the method `order(int v)`, which takes an `int`-value `v` and returns the value of the `ord`-array at the given index `v`. Its contract consists of one pre- and one postcondition. When the method is called, the parameter `v` is required to be in [`0`,`ord.length`), otherwise an `IndexOutOfBoundsException` would be thrown. The postcondition states that the value returned by `order(int v)` equals `ord[v]`. Also the `assignable`-clause says that no fields are modified in the method-body.

```
/*@ public normal_behavior
  @  requires 0 <= v && v < ord.length;
  @  ensures \result == ord[v];
  @  assignable \strictly_nothing;
  @*/
public int order(int v) {
  return ord[v];
}
```

Listing 2.3: Example of a method-contract

Additionally to describing a methods pre- and postconditions, it is possible to describe the behavior of loops within the method-body, by using loop-invariants (`loop_invariant`). For the loop to fulfill its specification, the loop-invariants must be preserved by the loop-body. Also the invariants must initially be true, before the loop is started. [LPC+08]

The `for`-loop in listing 2.4 sets the entries of the `ord`-array to `-1`. The loop's counter variable `t` is initially `0` and will then be increased until it reaches `ord.length`. That results in the first invariant, stating that `t` must be in [`0`,`ord.length`]. The second invariant states that in each cycle, the fields [`0`,`t`) of `ord` have already been set to `-1`. The assignable-clause here contains the counter-variable `t`. The `assignable`-clause states that all entries with an index in [`t`,`ord.length`) of `ord` may be assigned.

```
/*@ loop_invariant 0 <= t && t <= ord.length;
  @ loop_invariant (\forall int i; 0 <= i && i < t; ord[i] == -1);
  @ decreases ord.length - t;
  @ assignable ord[t .. (ord.length - 1)], t;
  @*/
for (int t = 0; t < ord.length; t++) {
  ord[t] = -1;
}
```

Listing 2.4: Specification of a for-loop using loop-invariants

When working with loops, it is often necessary to show their termination. To do so, the `decreases` keyword can be used. A `decreases`-clause consists of an `int` (or `long`) expression, which must be decreased by at least `1` after each cycle. Also the expression must not drop below `0`. In listing 2.4, the `decreases`-clause demands that `ord.length - t` decreases. Since `t` is increased by one each cycle and is at most `ord.length`, `ord.length - t` is decreasing and does not fall below `0`. Similar to loops, the termination of recursive methods can be proven, using a `measured_by` clause. It also contains an `int` expression which must strictly decrease with every recursive call.

There are several tools using the JML annotations, to check whether a program violates its specification. For example the KeY tool can be used for automated or interactive correctness proofs.

2.5 KeY

The KeY tool is an interactive theorem prover for Java Dynamic Logic. JavaDL is a dynamic logic for Java, a modal logic that can argue about programs. KeY takes Java source-code specified with JML and transforms it into Java Dynamic Logic proof-obligations. These proof-obligations are a set of JavaDL formulas that need to be proven, in order to show that the corresponding program is correct. KeY uses a sequent calculus to deduce the correctness of a given formula. While applying rules to the sequent, a proof-tree is built, whose leafs are open or closed proof-goals. The proof that is built is a tree, since KeY can apply rules with more than one premiss, which split the proof in different branches. An open proof-goal is a sequent, derived from the initial proof-obligation, that still hast to be proven. Both interactive and automatic verification are supported by KeY. During automatic proofs, built-in heuristics are used to complete the proof as far as possible. If there are still open goals left, the user can try to close the proof manually. [ABB+16]

KeY also supports the specification of nested code-blocks with block-contracts as well as an alternate technique to specify loops, so called loop-contracts (introduced to KeY in [Wac12] and [Lan18]).

Block-contracts allow to split a methods body into smaller code-blocks, specify their behavior and then use the contract as if they were method-contracts. The example in listing 2.5 shows a code-block, in which the values of the two variables `x` and `y` are swapped. Like a method-

contract, the blocks pre- and postconditions can be specified with `ensures`- and `requires`-predicates. To reference a variable's state just before the block, the `\before`-keyword can be used: `\before(x)`. In the example, it is used in the postcondition to state that after the block, `x` has the value of `y` before the block (and vice versa). The `\old`-keyword still references the surrounding methods prestate.

```
/*@ requires x == \old(x)};
  @ ensures x == \before(y);
  @ ensures y == \before(x);
  @ ensures y = \old(x);
  @ assignable x, y;
  @*/
{
 int z = x;
 x = y;
 y = z;
}
```

Listing 2.5: Example of a block-contract

Loop-contracts can be used for blocks, which start with a loop. By using a loop-contract, the loop can be specified with pre- and postconditions. Listing 2.6 shows an example of a loop-contract. In the `while`-loop, the entries of an array `a` are set to `0`. The precondition describes that `n` is in [`0`,`a.length`] and requires `a` not to be `null`. These preconditions must be true before every iteration. The `\before`-keyword references the state before the current iteration. Since the loop is seen as a tail-recursive procedure here, the `ensures`-predicate must be true after the current and all following iterations. In this way, the shown contract ensures that all entries of `a` with an index in [`\before(n)`,`a.length`) equal `0`.

```
/*@ requires 0 <= n && n <= a.length;
  @ requires a != null;
  @ ensures (\forall int i; \before(n) <= i && i < a.length; a[i] == 0);
  @ assignable a[n..a.length];
  @ decreases a.length - n;
  @*/
{
 while(n < a.length) {
   a[n] = 0;
   n++;
 }
}
```

Listing 2.6: Example of a loop-contract

Furthermore, KeY supports user-defined taclets. Taclets are a formalization of the rules used in KeY. It allows to define, when a rule is applicable and how KeY should treat the rule during automatic proofs. The example in listing 2.7, shows the *ifthenelse-true*-taclet. The rule is applicable on every *if-then-else* expression, whose guard is *true*. A taclet also defines, what happens, when the rule is applied. In the example, the *if*-expression with a *true* guard, is

replaced by the expression of the *then*-case. A KeY-user can write own taclets and apply them in any other proof, after the taclets are proven to be correct. [RU16]

```
ifthenelse-true { \find ( \if (true) \then (then) \else (else) )
                  \replacewith ( then )
};
```

Listing 2.7: Exemplary taclet treating true if-statements

2.6 Proof-Script-Debugger

The Proof Script Debugger is a tool, which allows editing and executing proof-scripts. With such a script, the execution of interactive steps on a proof can be automated. For one, a script may contain script-commands, which perform symbolic-execution, one-single-proofstep or the auto-mode of KeY. Additionally individual branches in the proof can be selected and specific rules can be applied to selected branches. In this way, interactively applied rules do not have to be applied again by hand, after a failed proof-attempt. Those are now applied during the execution of the script. [BGU17]

Listing 2.8 shows an exemplary script-snippet. The `symbex`-command completes the symbolic-execution, after which `tryclose` is performed on every open goal, trying to close it. The following `cases`-block allows to match branches, in order to apply rules or commands on open goals. In the example, there are two branches that can be matched. If the first branch is matched, then 'one single proofstep' is performed on the sequent. In the second case, the KeY-rule *cut* is applied: *cut* takes a formula and splits the branch into two: one assuming the formula is correct and one where the formula must be shown to be true. With these `cases`-blocks, those branches can be selected, where interactive steps were required in KeY and the necessary rules can be added to the case of a specific branch.

```
symbex;
forall{tryclose;}

cases {
  case match 'Pre (searchC) // if x_2 true // Normal Execution (x_arr != null) //.*':
    onestep;

  case match '#0 // #1 // if x_2 false // Normal Execution (x_arr != null) //.*':
    cut '1 + anonIn_k > 0';

  default :
}
```

Listing 2.8: Exemplary proof-script

3 Recursive Depth-First Traversal

In the following, we present the implementation, specification and verification of recursive depth-first traversal (see section 2.3 and algorithm 1). Since all three topics contain both data structure independent and data structure specific parts, we will first look at their similarities in sections 3.1 to 3.3. Sections 3.4 to 3.7 will discuss the data structure specific implementation-, specification- and verification-processes in detail. The complete source-files can be found in listings A.5 to A.10.

3.1 Implementation

The implementation of recursive DFT is based on Program 18.1 and Program 18.2 in [SeSc03]. It combines Program 18.1, which performs depth-first search, with an additional `for`-loop from 18.2, to achieve depth-first traversal. From this code-basis, we can remove the graph data structure (GraphDS), so we can insert those data structures, we want to compare (see section 2.2). In the following listings all data structure specific types and interactions appear as Java-comments and serve as placeholders for the actual graph data structure, which will be inserted in sections 3.4 to 3.7.

```
public class GraphDFT_recursive {
  private /*GraphDS*/ G;
  private int cnt;
  private int[] ord;

  public GraphDFT_recursive(/*GraphDS*/ GIn) {
    this.G = GIn;
    cnt = 0;
    ord = new int[G.size];

    for (int t = 0; t < G.size; t++) {
      ord[t] = -1;
    }
  }

  /*...*/
```

Listing 3.1: Class fields and constructor of the recursive DFT class

Listing 3.1 shows the class's fields `G`, `cnt` and `ord`. `G` is a placeholder for the used graph data structure. The integer `cnt` is used as a timer, to determine when a vertex is visited for the first time during DFT. Every time a yet unvisited vertex is visited, the current `cnt` value is stored in

the array `ord`, so at the end of DFT, `ord` contains the pre-order-indices of all vertices. Here a vertex is represented by an integer value that is an index of `ord`, so `ord[v]` is the pre-order-index of the vertex `v` and `ord.length` equals the number of vertices in `G`. The constructor sets `G` to the given graph `GIn` and `cnt` to `0`, since no verticecs have been visited yet. The `ord` array is created with a size of `G.size`, which is the number of vertices in `G`. All of `ord`'s entries are set to `-1`, denoting that all vertices are unvisited.

```
public void dft() {
  for (int k = 0; k < G.size; k++) {
    if (ord[k] == -1) {searchC(k);}
  }
}

public void searchC(int v) {
  ord[v] = cnt++;
  /*List*/ adj = /*G.adjacent(v)*/;

  for (/*int t : adj*/) {
    if (ord[t] == -1) {searchC(t);}
  }
}
```

Listing 3.2: Methods dft and searchC for performing depth-first traversal

Listing 3.2 contains the code of the `dft`- and `searchC`-method, which perform the actual depth-first traversal. The method `searchC` (`searchC` stands for 'search component') performs depth-first search and searches that component of the graph that is reachable from the given vertex. The `dft`-method calls `searchC` multiple times, which results in depth-first traversal.

The `searchC` method takes an integer, representing a vertex, and marks it as visited, by storing the current `cnt` value in `ord[v]`. That is now the pre-order-index of `v`. Also the `cnt` value is increased by one, since the next vertex to be visited, gets the next higher pre-order-index. After that, `searchC` is called recursively for every unvisited, adjacent vertex of `v`, which are stored in `adj`. Thereby, all vertices that are reachable from `v`, are visited after `searchC(v)` has finished. The retrieval and the type of `adj` depend on the used data structure, as well as the iteration of its elements.

The `dft`-method takes care that all of `G`'s vertices will be searched. It starts `searchC` with a vertex `k`. If not all vertices are reachable from `k`, some stay unvisited. Then `dft` calls `searchC` again for one of the remaining unvisited vertices and repeats this until all vertices are visited.

3.2 Specification

The specification of the implementation in section 3.1 contains some data structure specific invariants and pre-/postconditions in method-contracts. Still, there are some parts of the specification that remain the same, regardless of the used data structure. Those will be presented in the following, while the data structure specific parts are treated in sections 3.4

to 3.7. Until then, the specific specification parts are denoted by comments describing their purpose.

In this chapter, we will consider the algorithm to be correct, if it performs a graph-traversal on the whole graph, visiting each vertex exactly once. This specification is sufficient for us, to compare the used data structures (see chapter 6). A specification of the depth-first property is given in chapter 5.

The class-invariants, shown in listing 3.3, consist of the invariants concerning the used graph data structure. Additionally, there are invariants to the `cnt` field. Since it is counting the already visited vertices, it must be in [`0`,`G.size`], where `G.size` denotes the number of vertices in the graph. The `\num_of` quantifier sums up the number of fields `i` in `ord` where `ord[i] ≠ -1`, which represent the visited vertices. Thus, this sum always equals `cnt`. The `ord`-array stores the pre-order-index of each vertex. Since pre-order-indices must be disjoint, the fourth invariant is taking care that the `ord`-entries of visited vertices are a permutation of the values [`0`,`cnt`). Thus, after the `dft`-method terminates, every vertex has a unique pre-order-index. The specification also contains a model-method `unvisited()` (mentioned in listing 2.2), which returns those locations in `ord` that hold `-1`, representing an unvisited vertex. The vertex that will be visited next, depends on the graphs structure. So any location could be marked as visited, which is why there is no index in `ord`, which separates the visited vertices from the unvisited ones. Therefore, the model-method uses an *infinite_union* to collect those locations. This model-method can be used to describe which entries in `ord` are assignable in `dft` and `searchC`.

```
/*Data structure specific invariants*/
private /*@ spec_public @*/ /*GraphDS*/ G;

/*@ public invariant 0 <= cnt && cnt <= G.size;
  @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
  @*/
private /*@ spec_public @*/ int cnt;

/*@ public invariant ord.length == G.size;
  @ public invariant (\forall int i; 0 <= i && i < cnt;
                    (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @*/
private /*@ spec_public @*/ int[] ord;

/*@ model \locset unvisited() {
     return \infinite_union(int i;
          (0 <= i && i < ord.length && ord[i] == -1); \singleton(ord[i]));}
  @*/
```

Listing 3.3: Class-invariants of the recursive DFT class

The constructor (see listing 3.1) takes the graph data structure `GIn` and sets `G = GIn` and thus must assure that the input-graph `GIn` is valid. Therefore, the class's data structure-specific invariants are required of `GIn`. Also it ensures that `cnt` is set to `0` and that all entries of the

ord-array are set to -1. The specification of the loop, setting the ord-entries, is explained in listing 2.4.

```
/*@ public normal_behavior
  @ requires //Data structure specific requirement on GIn//
  @ ensures G == GIn;
  @ ensures (cnt == 0);
  @ ensures (\forall int i; 0 <= i && i < G.size; ord[i] == -1);
  @*/
public GraphDFT_recursive(boolean[][] GIn) {
  /*...*/
}
```

Listing 3.4: Specification of the constructor

The dft-method, responsible for calling searchC until all vertices are visited, is supposed to be called when no vertices have been visited. That is assured with the two requires-clauses in listing 3.5. Also the specification ensures that, after dft has finished, all vertices have been visited and the ord-array is a permutation of [0,G.size). The assignable-clause states that only the ord-array and cnt will be modified.

```
/*@ public normal_behavior
  @  requires cnt == 0;
  @  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
  @  ensures (cnt == G.size);
  @  ensures (\forall int i; 0 <= i && i < G.size;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @ assignable ord[*], cnt;
  @*/
public void dft() {
  /*...*/
}
```

Listing 3.5: Method-contract of dft

The specification of the loop in dft (listing 3.6) uses a loop-contract. It requires/ensures the class-invariants before/after each execution of the loop-body. Also, all vertices in [0,k) must be visited, where k is the loop's counter-variable. This is because in iteration k, either k is already visited or searchC(k) is called, which ensures that k is visited afterwards. After the loop has finished, the variable k equals G.size. Also cnt must be greater or equal to k, since at least those vertices in [0,k) have been visited. By using k = G.size and cnt $\geq$ k as well as the class-invariant cnt $\leq$ G.size, we can show that after the loop cnt equals G.size. During the loop, cnt and k may be assigned, as well as all unvisited vertices in ord (returned by unvisited()).

```
/*@ loop_contract normal_behavior
  @  requires \invariant_for(this);
  @  requires 0 <= k && k <= G.size;
  @  requires (\forall int i; 0 <= i && i < k; ord[i] != -1);
  @  requires cnt >= k;
  @  ...
  @  ensures (\forall int i; 0 <= i && i < k; ord[i] != -1);
  @  ensures cnt >= k;
  @  ensures \invariant_for(this);
  @  ensures k == G.size;
  @ decreases G.size - k;
  @ assignable unvisited(), cnt, k;
  @*/
{
  for (int k = 0; k < G.size; k++) {
    if (ord[k] == -1) {searchC(k);}
  }
}
```

Listing 3.6: Loop-contract of dft

The `searchC`-method in listing 3.7 recursively visits all reachable vertices, starting at the given vertex `v`. Thus the given integer must represent a vertex and must be in [`0`,`G.size`). Also `v` must not be visited yet. The postcondition ensures that `cnt` is increased by at least one, since at least `v` is visited. In order to prove that the `ord`-array is a permutation of [`0`,`G.size`), `searchC` has to ensure that (after `cnt - \old(cnt)` vertices have been visited) all timestamp-values in [`\old(cnt)`,`cnt`) were inserted into `ord`. Since `searchC` is a recursive method, its termination has to be shown with a `measured_by` clause. It states that with each call of `searchC` `cnt` is increased, so `G.size - cnt` decreases monotonically. Since `cnt` counts the visited vertices and is at most `G.size`, `G.size - cnt` never drops below `0`.

```
/*@ public normal_behavior
  @  requires 0 <= v && v < G.size;
  @  requires ord[v] == -1;
  @  ...
  @  ensures cnt > \old(cnt);
  @  ensures ord[v] == \old(cnt);
  @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @ measured_by (G.size - cnt);
  @ assignable unvisited(), cnt;
  @*/
public void searchC(int v) {
  /*...*/
}
```

Listing 3.7: Method-contract of searchC

The loop inside `searchC` is also specified by a loop-contract, shown in listing 3.8. Like the `dft`-loop it requires and ensures the class-invariants. Furthermore, it requires that `adj` is a list of `v`'s adjacent vertices in `G` (`v ∈ [0,G.size)`), which is used, to apply class-invariants on `adj`. The contract states as well that `v` remains visited before and after each iteration. It also ensures that `cnt` does not decrease and that only unvisited vertices are given a pre-order-index. Additionally the termination of `searchC` has to be shown. Since `cnt` is increased before the loop-contract-block and does not change inside it, it is required that `G.size` is the same before the method and the block. Also `cnt` before the method (`\old(cnt)`) has to be at least `0` and less than the `cnt` before the block. With these requirements, `\old(G.size - cnt)` before the method is larger than `G.size - cnt` when entering the block, which is also at least zero, since `cnt`'s invariants (see listing 3.3) must be true as well. With that, the loop can be shown to terminate.

```
/*@ loop_contract normal_behavior
  @  requires \invariant_for(this);
  @  requires adj == G.adjacent(v) && 0 <= v && v < G.size;
  @  requires ord[v] != -1 ;
  @  requires 0 <= \old(cnt) && \old(cnt) < cnt && \old(G.size) == G.size;
  @  ...
  @  ensures ord[v] != -1 && \before(cnt) <= cnt;
  @  ensures \invariant_for(this);
  @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
              (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @ measured_by G.size - cnt;
  @ decreases //adj.length - index//;
  @ assignable unvisited(), cnt, t;
  @*/
{
  for (/*int t : adj*/) {
    if (ord[t] == -1) {searchC(t);}
  }
}
```

Listing 3.8: Loop-contract of searchC

3.3 Verification-Process and Interactive-Steps

During the verification-process, we will encounter bounded sums (3.2) arising from the `\number_of`-quantifier in listing 3.3. We will have to use some properties of bounded sums, that are not yet built into KeY, in order to prove specific goals of the DFT-proof. Those properties appeared to be general ones, which might be reused in other proofs, when dealing with bounded sums. Since the bounded sums derive from a `\number_of`-quantifier, either 0 or 1 is added for each summed index. In the following, we will use the indicator function δ_{phi} (see (3.1)), which resolves to 1 if the formula *phi* is true and 0 otherwise:

$$\delta_{phi} = \begin{cases} 1 & \text{if } phi \\ 0 & \text{if } \neg phi \end{cases} \tag{3.1}$$

In (3.2) i_0 and i_2 are the lower and upper bound of the sum, not including i_2, so the summed range is $[i_0, i_2)$.

$$\sum_{i=i_0}^{<i_2} \delta_{phi(i)} \tag{3.2}$$

To receive reusable rules, we can define the following new KeY-taclets (see section 2.5):

- *bsum_num_of_bounds*:
  If a bounded sum like (3.2) can be found in the sequent, two statements can be added to the antecedents of the sequent. On the one hand, the bounded-sum results into a value of at least 0. On the other hand, it is at most the number of summed indices. With a range of $[i_0, i_2)$, the maximum value is $i_2 - i_0$. Still, this is only true, if $i_0 \leq i_2$. These properties are true for every lower and upper bound i_0 and i_2 as well as for every formula *phi*:

$$\forall i_0 : \forall i_2 : \forall phi : (0 \leq \sum_{i_0}^{<i_2} \delta_{phi(i)} \wedge i_0 \leq i_2 \rightarrow \sum_{i_0}^{<i_2} \delta_{phi(i)} \leq i_2 - i_0) \tag{3.3}$$

- *bsum_num_of_is_max*:
  If the bounded-sum equals $i_2 - i_0$, that means for each $i \in [i_0, i_2)$ a 1 is added, thus $phi(i)$ must be true for each i. That statement is added to the sequent:

$$\frac{\forall i : i_0 \leq i < i_2 \rightarrow phi(i), \quad \sum_{i=i_0}^{<i_2} \delta_{phi(i)} = i_2 - i_0 \implies}{\sum_{i=i_0}^{<i_2} \delta_{phi(i)} = i_2 - i_0 \implies} \tag{3.4}$$

  There is also a taclet *bsum_num_of_is_max2* that works the opposite way and adds the bounded-sum, if for each $i \in [i_0, i_2)$ $phi(i)$ is true.

- *bsum_num_of_lt_max*:
  This taclet can be applied, if a bounded-sum results into a value which is less than the maximum value (assuming that $i_0 < i_2$ and $0 \leq i_0$). In that case, there must be an $i \in [i_0, i_2)$, so that $phi(i)$ is false.

$$\frac{\exists i : (i_0 \leq i < i_2 \wedge \neg phi(i)), \quad 0 \leq i_0 \wedge i_0 < i_2, \quad \sum_{i=i_0}^{<i_2} \delta_{phi(i)} < i_2 - i_0 \implies}{0 \leq i_0 \wedge i_0 < i_2, \quad \sum_{i=i_0}^{<i_2} \delta_{phi(i)} < i_2 - i_0 \implies} \tag{3.5}$$

  This taclet also has a counterpart that works the other way: if the mentioned *exists*-statement is found, the bounded-sum can be added to the sequent.

These taclets have been proven to be correct in KeY. The full definition of the taclets can be found in listings A.1 to A.3 in the appendix.

Across all four data structures some goals had to be proven interactively. Some of these interactive parts remain the same for all of them. Regardless of the used data structure, there are three major points, where interactive steps are necessary, when proving recursive DFT.

The first one appears, when verifying the loop-contract inside the `dft`-method (see listing 3.6). In the case that $ord[k] \neq -1$ (which means that vertex `k` has already been visited), `searchC` is not called. Thus `cnt` (which counts the visited vertices) is not increased, while the loop's counting variable `k` is still increased by one. It has to be shown that the loops postcondition $cnt \geq k + 1$ is true. At first, we can show with KeY that for all $i \in [0, k + 1)$, $ord[i]$ does not equal -1:

$$\begin{aligned} &\sum_{i=0}^{<G.size} \delta(ord[i] \neq -1) = cnt, \\ &\forall 0 \leq i < k + 1 : ord[i] \neq -1 \implies cnt \geq k + 1 \end{aligned} \tag{3.6}$$

In (3.6), we see the current open goal, where the left side of the sequent shows the assumptions and the right side shows the postcondition we want to prove. It contains the formula, we have just sown with KeY and the class-invariant (see section 3.2), which states, how *cnt* equals a bounded sum. We can apply the aforementioned taclet *bsum_num_of_is_max2* (see (3.4)) on the universal quantifier in (3.6), which states that the following is true:

$$(3.6) \overset{(3.4)}{\implies} \sum_{i=0}^{<k+1} \delta(ord[i] \neq -1) = k + 1 - 0 \tag{3.7}$$

The sum in (3.6) can be split (*bsum_split*-rule) at $k + 1$ into two sums:

$$(3.6) \implies \underbrace{\sum_{i=0}^{<k+1} \delta(ord[i] \neq -1)}_{\overset{(3.7)}{=} k+1} + \underbrace{\sum_{i=k+1}^{<G.size} \delta(ord[i] \neq -1)}_{\overset{(3.3)}{\geq} 0} \overset{(3.6)}{=} cnt \tag{3.8}$$

The first sum in (3.8) equals `k+1` as shown in (3.7) and the second one can be shown to be at least `0`, using the *bsum_num_of_bounds*-taclet (3.3). Together that concludes in *cnt* being at least $k + 1$.

While proving the loop-contract in `dft`, we have to show that `searchC`'s preconditions (see listing 3.7) are valid, in case $ord[k] = -1$. In that case `searchC` will be called with k, which will start DFS at vertex k, thus there is at least one unvisited vertex. Here, proving that *cnt* is less than *G.size*, requires interactive steps. Since *cnt* must be less or equal to *G.size*, we can first

assume that they are equal and show that it resolves into a contradiction:

$$\begin{aligned} cnt = G.size - 0,\\ \sum_{i=0}^{<G.size} \delta(ord[i] \neq -1) = cnt\\ ord[k] = -1,\\ cnt \leq G.size, \implies cnt < G.size \end{aligned} \tag{3.9}$$

In (3.9), we see the current open goal with the assumptions on the sequent's left sine and precondition we want to prove on its right side. It contains the mentioned assumption and established class-invariants, which state that *cnt* equals a bounded sum and that it is at most *G.size*. By applying the assumed equality on *cnt* in the first invariant, we can deduce the following:

$$(3.9) \implies \sum_{i=0}^{<G.size} \delta(ord[i] \neq -1) \overset{(3.9)}{=} G.size - 0 \tag{3.10}$$

As the sum in (3.10) equals the number of summed indices, we can apply the taclet *bsum_num_of_is_max* (3.4), which results in the following:

$$(3.10) \overset{(3.4)}{\implies} \forall i : (0 \leq i < G.size \rightarrow \delta(ord[i] \neq -1)) \tag{3.11}$$

Still, (3.11) cannot be true, since $ord[k] = -1$.

These interactive steps are also required when proving `searchC`'s precondition in `searchC`'s inner loop-contract.

The third interactive point appears, when proving the `searchC`-method-contract. The method `searchC` takes the vertex v, from where DFS will start. After v is marked as visited and *cnt* is increased by one, the class-invariants must be shown to be valid, since they are required by the following loop-contract. As *cnt* and *ord* are modified, there will be new notations for *cnt* and *ord* in the following: cnt_b and ord_b denote *cnt* and *ord* before the execution of `searchC`'s body, and cnt_{in} and ord_{in} denote *cnt*'s and *ord*'s value just before entering the loop-contract-block. Since v is marked as visited before the loop-contract-block is entered and was unvisited before the block, $ord_b[v]$ is -1 and $ord_{in}[v]$ equals cnt_b. Also *cnt* is increased by one, hence cnt_{in} equals $cnt_b + 1$.

$$\begin{aligned} ord_b[v] = -1,\\ ord_{in}[v] = cnt_b,\\ cnt_{in} = cnt_b + 1,\\ \sum_{i=0}^{<G.size} \delta(ord_b[i] \neq -1) = cnt_b \implies \sum_{i=0}^{<G.size} \delta(ord_{in}[i] \neq -1) = cnt_{in} \end{aligned} \tag{3.12}$$

In (3.12), the open goal that has to be proven, is shown: the sequent's left side contains the goal's assumptions, while the right side contains the invariant we want to prove. The

sums in (3.12) must be equal to cnt before (cnt_b) and after it is increased (cnt_{in}), since the class-invariants (see section 3.2) are true before the method and must still be true before the block. At first, we can split both sums in (3.12) at the index (vertex) v into three parts (KeY's *bsum_split_in_three*-rule):

$$\begin{aligned}\sum_{i=0}^{<v} \delta(ord_b[i] \neq -1) + \delta(ord_b[v] \neq -1) + \sum_{i=v+1}^{<G.size} \delta(ord_b[i] \neq -1) = cnt_b \implies \\ \sum_{i=0}^{<v} \delta(ord_{in}[i] \neq -1) + \delta(ord_{in}[v] \neq -1) + \sum_{i=v+1}^{<G.size} \delta(ord_{in}[i] \neq -1) = cnt_{in}\end{aligned} \tag{3.13}$$

The arrays ord_b and ord_{in} only differ at the index v, since the vertex v is unvisited in ord_b and marked as visited in ord_{in}. Thus we can prove that both sums with the range $[0, v)$ in (3.13) are equal and so are both sums with the range $[v + 1, G.size)$:

$$\sum_{i=0}^{<v} \delta(ord_b[i] \neq -1) = \sum_{i=0}^{<v} \delta(ord_{in}[i] \neq -1) \land \sum_{i=v+1}^{<G.size} \delta(ord_b[i] \neq -1) = \sum_{i=v+1}^{<G.size} \delta(ord_{in}[i] \neq -1) \tag{3.14}$$

With the *equal_bsum*-rule we can prove that (3.14) is true. By applying both equalities in (3.14) on (3.13), we can deduce the following:

$$(3.13) \overset{(3.14)}{\implies} cnt_b - \delta(ord_b[v] \neq -1) + \delta(ord_{in}[v] \neq -1) = cnt_{in} \tag{3.15}$$

Since v is marked as visited before the loop-contract-block is entered and was unvisited before the block, $\delta(ord_{in}[v] \neq -1)$ is 1 and $\delta(ord_b[v] \neq -1)$ is 0. Also cnt is increased by one, hence cnt_{in} equals $cnt_b + 1$. With that the (3.16) can be resolved, which is true.

$$(3.15) \overset{(3.12)}{\implies} cnt_b - 0 + 1 = cnt_b + 1 \tag{3.16}$$

For recursive DFT with adjacency matrix, proof-scripts (see section 2.6) were created, which contain those rules that had to be applied manually. After executing those scripts, a few goals were left open. These could then be closed manually in KeY. Still a significant number of interactions could be omitted: originally 101 interactive steps were used to close these proofs. Using the scripts, reduced the number to 14 interactive steps.

3.4 Adjacency Matrix

Adjacency matrices are implemented as two-dimensional `boolean`-arrays. The array-indices represent the graphs vertices and `G[i][j]` describes, whether there is an edge (i, j) in `G`. Listing 3.9 shows the properties of the matrix, including that it has to be a square matrix, and that there is at least one vertex.

```
/*@ public invariant (\forall int j; 0 <= j && j < G.length;
                      G[j] != null && G[j].length == G.length);
  @ public invariant G.length > 0;
  @*/
private /*@ spec_public @*/ boolean[][] G;
```

Listing 3.9: Implementation and specification of the adjacency matrix

In order for the constructor to establish the class-invariants, these properties are also required of the constructors input `GIn`, as shown in listing 3.10.

```
/*@ public normal_behavior
  @ requires (\forall int j; 0 <= j && j < GIn.length; GIn[j].length == GIn.length);
  @ requires GIn.length > 0;
  @ ...
  @*/
public GraphDFT_matr(boolean[][] GIn) {
  /*...*/
}
```

Listing 3.10: Constructor setting the adjacency matrix

Another adjacency matrix specific implementation is the retrieval of a vertex's adjacent vertices. In `searchC` the variable `adj` is assigned `G[v]`, where `v` is the currently visited vertex. In order to use class-invariants concerning `G[v]`, to prove properties of the `adj`-array, the information of `adj = G[v]` must also be passed to the loop-contract. `G[v]` does not actually contain `v`'s adjacent vertices, but rather the information for each vertex, whether it is adjacent to `v`. Thus, for each vertex `t` in `G`, additionally to `ord[t]` being `-1`, `adj[t]` must be true, before `searchC(t)` can be called. Thus `adj[t]` is added to the if-guard inside the `searchC`-block, as listing 3.11 shows.

```
public void searchC(int v) {
  ord[v] = cnt++;

  boolean[] adj = G[v];

  /*@ loop_contract normal_behavior
    @  requires adj == G[v] && 0 <= v && v < G.length;
    @  ...
    @ decreases adj.length - t;
    @*/
  {
    for (int t = 0; t < adj.length; t++) {
      if (adj[t] && ord[t] == -1) {searchC(t);}
    }
  }

}
```

Listing 3.11: Retrieval of adjacent vertices with the adjacency matrix

Except those parts that need to be proven interactively across all data structures (see section 3.3), there are no additional interactions needed. Since these mentioned steps still have to be applied manually in KeY, we can use the Proof Script Debugger with a proof-script, which automates most of these steps (see section 3.3).

3.5 Adjacency Array

The adjacency array representation (see section 2.2.3) is a compressed form of an adjacency matrix. It can be implemented, using two `int`-arrays `G` and `adjArr` instead of a two-dimensional array-matrix. The array `adjArr` holds the concatenation of all vertices' adjacency lists. Because of the dummy-entry in `G`, the number of vertices in the graph is not `G.length` like for the adjacency matrix, but `G.length-1`. The specification contains a model-field `len` representing that number of vertices, which enables a more readable specification. Listing 3.12 also shows the class-invariants concerning the two mentioned arrays. For one, `G`'s values are indices to `adjArr` and therefore must be in `adjArr`'s bounds [`0`,`adjArr.length`) (this must also be true for the dummy entry `G[len]`). Also the referenced sections [`G[j]`,`G[j+1]`), which contain a vertex `j`'s adjacent vertices, must be disjoint for each vertex `j`. Hence `G`'s values must grow monotonically, as the index increases. Also, `len` must be greater than `0`, for the graph should contain at least one vertex. The invariant of `adjArr` states that its values must be in [`0`,`len`), since the entries describe the end-vertex of an edge. Furthermore `G`, `adjArr` and `ord` are all `int`-arrays. The third class-invariant states that those shall never reference the same array. This prevents aliasing problems during the verification.

```
/*@ public invariant (\forall int j; 0 <= j && j < len;
                      0 <= G[j] && G[j] < adjArr.length && (G[j] <= G[j+1]));
  @ public invariant len > 0;
  @ public invariant G != adjArr && adjArr != ord && ord != G;
  @ public invariant 0 <= G[len] && G[len] < adjArr.length;
  @*/
private /*@ spec_public @*/ int[] G;

/*@ model int len;
  @ represents len == G.length - 1;
  @*/

/*@ public invariant (\forall int i; 0 <= i && i < adjArr.length;
                      0 <= adjArr[i] && adjArr[i] < len);
  @*/
private /*@ spec_public @*/ int[] adjArr;
```

Listing 3.12: Implementation and specification of the adjacency array

The specification of the constructor, which takes two `int`-arrays `GIn` and `adjIn`, states that those must not reference the same array. It also requires the rest of the aforementioned invariants of `GIn` and `adjIn`. Also, due to the dummy entry in `G`, the `ord`-array is instantiated with an array of size `G.length - 1`, as listing 3.13 shows.

```
/*@ public normal_behavior
  @ requires GIn.length - 1 > 0;
  @ ...
  @ requires GIn != adjIn;
  @ requires 0 <= GIn[GIn.length - 1] && GIn[GIn.length - 1] < adjIn.length;
  @*/
public GraphDFT_adjarr(int[] GIn, int[] adjIn) {
  this.G = GIn;
  this.adjArr = adjIn;
  ord = new int[G.length - 1];
```

Listing 3.13: Constructor setting the adjacency array

Since `adjArr` contains the adjacent vertices for each vertex, there is no need to retrieve a list or an array which only contains `v`'s adjacent neighbors. Instead, the loop's bounds can be set to those stored in `G[v]` and `G[v+1]` dynamically, so the counting-variable `i` only iterates `v`'s neighbors (see listing 3.14).

```
public void searchC(int v) {
  /*...*/

  for (int t = G[v]; t < G[v+1]; t++) {
    if (ord[adjArr[t]] == -1) {searchC(adjArr[t]);}
  }
```

Listing 3.14: Iterating adjacent vertices with the adjacency array

With the loop behaving differently, its specification also differs form the one used for the adjacency matrix. For one, `t` must be in `[G[v],G[v+1])` and thus the `decreases`-clause requires `G[v+1] - t` to decrease. Also `v` must be in `[0,len)`. This is needed, so we can apply class-invariants concerning `G[v]` and `G[v+1]`, since those invariants are only true for vertices in these bounds.

```
/*@ loop_contract normal_behavior
  @  requires G[v] <= t && t <= G[v+1];
  @  requires 0 <= v && v < len;
  @  ...
  @ decreases G[v+1] - t;
  @*/
```

Listing 3.15: Loop-contract of searchC with adjacency array

Apart from those interactive steps that all data structures require, the verification of recursive DFT with adjacency array needs some more interactive steps. For example proving that the invariant $\forall i : (0 \leq i < ord.length \rightarrow \exists j : (0 \leq j < ord.length \wedge ord[j] = i))$ remains valid after the `dft`-method, when using the loop-contract (which ensures the class-invariants). Here it is sufficient to skolemize the invariant's instance we want to prove (on the sequent's right side), which creates a new skolem-constant i_0: $(0 \leq i\_0 < ord.length \rightarrow \exists j : (0 \leq j < ord.length \wedge ord[j] = i\_0))$. After that, the appearances of i in the established instance of that invariant (on the sequent's left side) can be instantiated with i_0 and KeY can close the goal by itself. These steps are applied at several points throughout the adjacency array proofs, where this invariant appears. Also proving the `assignable`-clause in `dft`'s and `searchC`'s inner loop-contract, when using `searchC`'s method-contract, requires interaction. Both `assignable`-clauses contain `unvisited()`, which returns those locations in the `ord`-array that hold `-1`. Here, the definition of `unvisited` has to be unrolled on both, the sequent's left and right side. Then, after skolemizing the resolving existential quantified formula on the left and instantiating the one on the right, KeY can close the goals.

3.6 Linked Data Structure

The linked data structure models vertices as `Vertex`-objects that hold a `Vertex`-array `adj` of adjacent vertices (shown in listing 3.16). Additionally each Vertex-object has an `int`-id `val`, which is later used as index of the `ord`-array. Thus `val` must be at least `0`. The rest of `Vertex`'s class-invariants state that no entry of `adj` may be `null` and that all entries are pairwise distinct. The latter ensures that there are no multiple edges $e_1 = e_2 = (u, v) \in E$. Furthermore, every

vertex-object appears only once in the `adj`-array and has a `val`-id, which is used as index in the graph-array `G` later (see listing 3.17). The class is declared final, since that allows the `Vertex`'s invariants to be unrolled on the sequent's right side during the verification.

```
public final class Vertex {

  /*@ public invariant val >= 0; @*/
  int val;

  /*@ public invariant adj != null;
    @ public invariant (\forall int i; 0 <= i && i < adj.length; adj[i] != null);
    @ public invariant (\forall int i; 0 <= i && i < adj.length;
                (\forall int j; 0 <= j && j < adj.length;
                  (i==j) || (adj[i] != adj[j] && adj[i].val != adj[j].val)));
    @*/
  Vertex[] adj;
}
```

Listing 3.16: Implementation and specification of the Vertex-class

The graph, used by the DFT-algorithm, is implemented as an array `G` of `Vertex`-objects, which entails similar class-invariants as the `Vertex`-class. For example pairwise distinctness of the stored `Vertex`-objects is required as well. Additionally, for each `i` $\in$ [`0`,`G.length`) the `Vertex` at `G[i]` is supposed to have the `val`-id `i`. This also ensures that the `val`-ids of the graphs vertices are unique. The specification in listing 3.17 also contains that for each `Vertex` in `G`, its class-invariants must hold.

```
/*@ public invariant (\forall int i; 0 <= i && i < G.length; G[i].val == i);
  @ ...
  @ public invariant (\forall int i; 0 <= i && i < G.length;
                  (\forall int j; 0 <= j && j < G.length;
                     (i!=j) ==> (G[i] != G[j]) ));
  @ public invariant (\forall int i; 0 <= i && i < G.length; \invariant_for(G[i]));
  @*/
private /*@ spec_public @*/ Vertex[] G;
```

Listing 3.17: Implementation and specification of the linked data structure

These invariants are then required of the constructor's argument `GIn` (see listing 3.18), like they are for the other data structures. Especially it requires for each `Vertex` in `GIn` that its class-invariants hold. This is needed, to establish `GraphDFT_linked`'s invariant, which states that they hold for each `Vertex` in `G`.

```
/*@ public normal_behavior
  @ requires (\forall int i; 0 <= i && i < GIn.length; GIn[i].val == i);
  @ ...
  @ requires (\forall int i; 0 <= i && i < GIn.length; \invariant_for(GIn[i]));
  @ ...
  @*/
public GraphDFT_linked(Vertex[] GIn) {
```

Listing 3.18: Constructor setting the linked data structure

With a different data structure, the retrieval of a vertex's adjacent vertices changes as well. Listing 3.19 shows how the local `Vertex`-array `adj` is assigned the current vertex's array `G[v].adj`. The following loop then iterates the `Vertex`-objects in `adj`. To get the `Vertex`'s index in `ord`, the `adj[t].val`-id is used, which corresponds to the `Vertex`'s position in `G`.

```
public void searchC(int v) {
  /*...*/
   Vertex[] adj = G[v].adj;

  for (int t = 0; t < adj.length; t++) {
    if (ord[adj[t].val] == -1) {searchC(adj[t].val);}
  }
```

Listing 3.19: Retrieval of adjacent vertices with the linked DS

The loop's specification is mostly the same as the one for the adjacency matrix. Still, some requirements are added: for example `v` must be in [`0`,`G.length`) as it does with the adjacency array data structure (see listing 3.15). The third requirement in listing 3.20 makes sure that the `adj`-array equals `G[v].adj`, which is not known inside the block contract, since `adj` is created outside of it. With that, `G[v].adj`'s properties can be applied to `adj`, since they are contained in the class-invariants (which are also required in the loops preconditions).

```
/*@ loop_contract normal_behavior
  @  requires \invariant_for(this);
  @  requires 0 <= v && v < G.length;
  @  requires adj == G[v].adj;
  @  ...
  @*/
```

Listing 3.20: Specification of searchC's loop-contract with the linked DS

The verification of this DFT version requires interactive assistance for proving the `assignable`-clause in `dft`'s inner loop-contract, like the adjacency array proof does (see section 3.5). Apart from that, each time the class's invariants must be proven to hold, there is one of them that requires some interactions. That invariant is the third one in listing 3.17, which states that the class-invariants of all `Vertex`-objects in `G` are fulfilled. The sequent contains an instance of this invariant on both its left and right side, which both correspond to different heap-states. Still these different heaps do not vary with respect to the graph data structure, since that always remains untouched. After skolemizing the right instance and instantiating the left one with the newly created skolem-constant, the class-invariants of `Vertex` can be unrolled. The different

invariants of `Vertex` can then be split into separate branches, which KeY can close by itself, except one invariant, which contains two nested universal-quantifiers. That branch can be closed by skolemizing and instantiating those quantifiers manually.

3.7 Adjacency Lists

The last version of recursive DFT uses adjacency lists as graph data structure. Here the graph is implemented as an array containing `List`-objects, which hold indices of adjacent vertices. The invariants in listing 3.21 make sure that the `Lists`'s elements are values in [`0`,`G.length`). Also each list may not be longer than the graphs number of vertices `G.length` and its invariants must be true. As list we use a `List`-interface provided by KeY's examples [1], which is simplified for our needs, so it only allows `int`-values (see listing A.10). This interface provides a `size`-method which returns the number of elements in the `List` and a `get(int i)`-method, which returns the element at the given position of the `List`. Since the elements of a lists, are stored at arbitrary locations on the heap, the last invariant makes sure that they do not alias with the DFT-class's fields. The `footprint` of a `List`, is a collection of its elements locations. While performing DFT, the `cnt`-field and `ord`'s entries are manipulated. Hence the last invariant states that the `footprint` of each list is disjoint to the location of `cnt` as well as to the locations of `ord`'s entries. This is needed to prove that assigning `cnt` or `ord`'s entries does not change the graphs structure.

```
/*@ public invariant (\forall int j; 0 <= j && j < G.length;
                        G[j] != null && G[j].size() <= G.length);
  @ public invariant G.length > 0;
  @ public invariant (\forall int i; 0 <= i && i < G.length;
                        (\forall int j; 0 <= j && j < G[i].size();
                            0 <= G[i].get(j) && G[i].get(j) < G.length));
  @ public invariant (\forall int i; 0 <= i && i < G.length; \invariant_for(G[i]));
  @ public invariant (\forall int i; 0 <= i && i < G.length;
                       \disjoint(G[i].footprint,\singleton(cnt))
                    && (\forall int j; 0 <= j && j < ord.length;
                        \disjoint(G[i].footprint,\singleton(ord[j])) ));
  @*/
private /*@ spec_public @*/ List[] G;
```

Listing 3.21: Implementation and specification of adjacency lists

Listing 3.22 shows how the adjacent vertices of the current vertex `v` are obtained: here the list `adj` is simply assigned the list in `G[v]`. This resembles the way it was retrieved with the linked data structure (see section 3.6), where `adj` is assigned the array stored in the `Vertex` `G[v]`. Thus their invariants are similar as well, except for the access to an adjacent vertex in the list or its length, which is now accomplished with `get()` and `size()`. Here, `adj` must equal `G[v]`, so the properties concerning that list, which are contained in the class-invariants, can be used for `adj` in the block as well.

[1] Dynamic Frames/List with Sequences/List.java

```
/*@ loop_contract normal_behavior
  @  requires adj == G[v];
  @  requires 0 <= v && v < G.length;
  @  ...
  @*/
public void searchC(int v) {
  ord[v] = cnt++;
  List adj = G[v];

  for (int t = 0; t < adj.size(); t++) {
    if (ord[adj.get(t)] == -1) {searchC(adj.get(t));}
  }
```

Listing 3.22: Retrieval of adjacent vertices with adjacency lists

The verification of recursive DFT with adjacency lists requires interactions at several points. Similar to the verification with the adjacency array (see section 3.5), proving the assignable-clauses in `searchC`'s and `dft`'s loop-contract, when using `searchC`'s method contract, requires interactive steps. Also similar are the interactions needed for the invariant $\forall i : (0 \leq i < ord.length \rightarrow \exists j : (0 \leq j < ord.length \wedge ord[j] = i))$. Furthermore, proving that `adj.get(t)` is in the range of `[0,G.length)` in `searchC`'s loop-contract requires interactive help. The second invariant in listing 3.21 states that all entries j in all lists i in G, are in that range. By instantiating i with v (since the precondition provides `adj = G[v]`) and j with t, $0 \leq G[v].get(t) \wedge G[v].get(t) < G.length$ can be deduced. Also, unrolling the queries `.get()` and `.size()` required some attention, when proving their preconditions to be true.

4 Non-Recursive Depth-First Traversal

In this chapter we will discuss the implementation, specification and verification of the non-recursive form of DFT. We will look at these processes for all four data structures, as we did with the recursive form in chapter 3. This provides another proof-example, to compare the data structures (see section 6.4). Also, we can compare the recursive and non-recursive form of DFT, regardless of the used data structure (see section 6.3). In the following, we will first discuss the data structure independent parts of the implementation (section 4.1), specification (section 4.2) and verification (section 4.3). The specifics of each data structure are presented in sections 4.4 to 4.7. The complete source-files can be found in listings A.11 to A.14.

4.1 Implementation

The implementation of non-recursive DFT is based on the pseudo-code in [Lar15]. For the recursive and the non-recursive form of DFT to be comparable, we will add the `cnt`-field and the `ord`-array, as well as the code to store the time-stamp in the ord-array, when a vertex is visited. In this way, both DFT-forms produce an array containing the pre-order-index of each vertex. The following listings show the data structure independent implementation of non-recursive DFT, thus all appearances of a specific data structure are replaced by a Java-comment. These placeholders are later replaced by one of the compared data structures (see section 2.2).

```
public class GraphDFT_nonrec {
  /*GraphDS*/ G;
  int cnt;
  int[] ord;
  int stackPtr;
  int[] stack;
```

Listing 4.1: Class fields of the non-recursive DFT class

The non-recursive form of DFT uses an explicit stack, on which visited but not yet backtracked vertices are stored. As listing 4.1 shows, the stack is implemented as a `stack`-array of sufficient size, accompanied by a `stackPtr`-integer, which points to the first empty field in the `stack`-array. Originally an external `Stack` class was used. However, already a minimum example of pushing and popping one `int` onto and off the stack rose many difficulties. Since we want to compare the graph data structures and not the stack data structure, we will use the easier implementation using an array. As in the recursive implementation (see listing 3.1), the `cnt`-field serves as time-stamp, which will be increased every time a new vertex is visited. Also the `ord`-array reappears, to store these time-stamps. Here again, the entry `ord[v] = -1` implies that the vertex `v` has not been visited.

```
public GraphDFT_nonrec(/*GraphDS*/ GIn) {
  G = GIn;
  cnt = 0;
  ord = new int[G.size];
  stackPtr = 0;
  stack = new int[G.size * G.size];

  for(int i = 0; i < ord.length; i++) {
    ord[i] = -1;
  }

  for(int k = 0; k < stack.length; k++) {
    stack[k] = -1;
  }
}
```

Listing 4.2: Constructor of the non-recursive DFT class

The mentioned fields are set in the constructor (see listing 4.2). Since there are no vertices visited at first, `cnt` is initially set to `0` and all entries in `ord` are set to `-1`. Also the stack is empty at first, so the `stackPtr` points to the field with index `0` in the `stack`-array and all `stack`-entries are set to `-1`. The `stack`-array's size is the number of vertices in `G` squared, which is a rough upper bound to the maximum of vertices that will be on the stack at the same time. Why this size is sufficient, will be discussed later in this section.

```
public void dft() {
  for(int k = 0; k < G.size; k++) {
    if (ord[k] == -1){searchC(k);}
  }
}
```

Listing 4.3: The dft-method of the non-recursive DFT class

The implementation of the `dft`-method does not differ from the recursive one (see listing 3.2). Here as well, it takes care that all vertices will be searched and calls `searchC` for each vertex that has not been visited during the previous call.

The non-recursive implementation of the `searchC`-method contains the core difference to the recursive implementation. Here `searchC` does not call itself recursively, but uses the stack instead.

```
public void searchC(int v) {
  stack[stackPtr++] = v;

  while(stackPtr != 0) {
    int u = stack[--stackPtr];
    stack[stackPtr] = -1;

    if (ord[u] == -1) {
      ord[u] = cnt++;

      /*List*/ adj = /*G.adjacent(u)*/;

      for(/*int t : adj*/) {
        if (ord[t] == -1) {
          stack[stackPtr++] = t;
        }
      }

    }
  }
}
```

Listing 4.4: The searchC-method of the non-recursive DFT class

As shown in listing 4.3, `searchC` takes an integer `v`, which is the vertex, where depth-first search will start. So `v` is put on the stack (which should be empty before that) and the `stackPtr` is increased by one, so it points to the first empty field in the `stack`-array. After that a `while`-loop starts, which will repeat its body until the stack is empty, which is the case if `stackPtr` equals `0`. In each iteration, the vertex on top of the stack is removed from it and stored as local varable `u`. In case `u` has already been visited, the loop continues with the next iteration or stops. Otherwise `u` is marked as visited, by storing the current time-stamp `cnt` at `ord[u]`. After that `cnt` is increased by one, as there is one more visited vertex now. Then the adjacent vertices of `u` are stored in `adj`. The type and the retrieval of `u`'s adjacency-list depend on the used data structure and so does the iteration of the elements in said list. In the following loop, all unvisited vertices that are adjacent to `u` are pushed onto the stack. In this way, the next vertex to be visited is one of the last visited vertex's unvisited children (if any exist). If no unvisited children are pushed onto the stack, all visited vertices are popped off the stack, until an unvisited one is on top. At this point, it is possible to estimate a maximum size of the stack. When pushing unvisited vertices in `adj` on the stack, at most `G.size` elements are iterated, which is the number of vertices in `G`. If for each vertex `G.size` vertices were pushed onto the stack, this would result into a stack of the size `G.size * G.size`. This is a rough upper bound, since there always is a vertex that is popped off the stack, before new vertices are pushed onto it. Also, there are less unvisited vertices each iteration, since one is always marked as visited before new vertices are pushed onto the stack. Still, this rough upper bound suffices to verify non-recursive DFT.

4.2 Specification

The implementation of non-recursive DFT (see section 4.1) has many properties that can be specified without knowledge of the used data structure. These will be presented in the following, while the specific specification of each data structure is discussed in sections 4.4 to 4.7.

Apart from the data structure's invariants, the class-invariants in listing 4.5 contain the same properties of `cnt` and `ord`, as they do in the recursive form (see listing 3.3). The `cnt`-field must still be in `[0,G.size]` and equals the number of entries in `ord` that are not `-1`. Also `ord`'s size must equal the number of vertices in the graph `G.size` and it must contain a permutation of the visited vertices. Since the non-recursive implementation uses a stack, there are some invariants concerning the `stackPtr`-field and the `stack`-array (see listing 4.6). The `stackPtr` holds an index to the `stack`-array, which is why it can be at least `0` (in case the stack is empty) and at most `stack.length` (in case the stack is full). Additionally the `stackPtr` is always less or equal to `cnt + 1` times the number of vertices in `G`. This is because after a vertex is visited `cnt` is increased by one and at most `G.size` vertices are pushed onto the stack, thus `stackPtr` is increased by at most `G.size`. The `+ 1` in `cnt + 1` is necessary, since starting `starchC(v)` puts `v` on the stack while `cnt` is still `0`: at this point `stackPtr = 1` $\leq$ `(0 + 1) * G.size = (cnt + 1) * G.size` is true but `stackPtr = 1` $\leq$ `0 = 0 * G.size = cnt * G.size` is wrong. This invariant will be used later on, to prove that the `stackPtr` stays in bounds. The invariants of the `stack`-array state that `stack`'s size equals the number of vertices in `G` squared. Also, to avoid aliasing, the invariants state that `ord` does not equal `stack`. Finally, the entries of `stack` need to be restricted: for one, the entries in `[0,stackPtr)` hold vertices that were pushed on the stack, and thus must be values in `[0,G.size)`. All other entries in `[stackPtr,stack.length)` must be `-1`. Apart from the mentioned fields there are two additional ghost-fields: `before` and `diff` that will appear in the `searchC`-method.

```
public class GraphDFT_nonrec {
  /*Data structure specific invariants*/
  /*GraphDS*/ G;

  /*@ public invariant 0 <= cnt && cnt <= G.size;
    @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
    @*/
  int cnt;

  /*@ public invariant ord.length == G.size;
    @ public invariant (\forall int i; 0 <= i && i < cnt;
                        (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @*/
  int[] ord;
```

Listing 4.5: Class-invariants of the non-recursive DFT class

```
  /*@ public invariant 0 <= stackPtr && stackPtr <= stack.length
                            && stackPtr <= (cnt + 1)*G.size;
    @*/
  int stackPtr;

  /*@ public invariant stack.length == (G.size * G.size);
    @ public invariant (\forall int i; 0 <= i && i < stackPtr;
                         0 <= stack[i] && stack[i] < G.size);
    @ public invariant (\forall int i; stackPtr <= i && i < stack.length;
                          stack[i] == -1);
    @ public invariant stack != ord;
    @*/
  int[] stack;

  /*@ ghost int diff;
    @ ghost int before;
    @*/
```

Listing 4.6: Class-invariants of the non-recursive DFT class concerning the stack

These invariants are established by the constructor, which does ensure some additional properties, as shown in listing 4.7. First of, it ensures that `G` is set to the given parameter `GIn`. Also, `stackPtr` and `cnt` are ensured to be `0`. Finally, the constructor ensures that all entries in the `ord`- and `stack`-array are set to `-1`.

```
/*@ public normal_behavior
  @ requires //Data structure specific requirement on GIn//
  @ ensures G == GIn;
  @ ensures stackPtr == 0;
  @ ensures (cnt == 0);
  @ ensures (\forall int j; 0 <= j && j < ord.length; ord[j] == -1);
  @ ensures (\forall int j; 0 <= j && j < stack.length; stack[j] == -1);
  @*/
public GraphDFT_nonrec(/*GraphDS*/ GIn) {
  /*...*/
}
```

Listing 4.7: Specification of the constructor

As the non-recursive implementation of the `dft`-method does not differ from the recursive one (see listings 3.2 and 4.3), its specification is mostly the same as well (see listing 3.5). The method-contracts match, except the `assignable`-clause, which here also contains `before`, `diff`, `stackPtr` and all entries of `stack`. Also it requires that `stackPtr` is `0`, since no vertices should be visited before `dft` is called.

```
/*@ public normal_behavior
  @  requires cnt == 0 && stackPtr == 0;
  @  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
  @
  @  ensures (cnt == G.size);
  @  ensures (\forall int i; 0 <= i && i < G.size; ord[i] != -1);
  @  ensures (\forall int i; 0 <= i && i < G.size;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @ assignable ord[*], cnt, before, diff, stack[*], stackPtr;
  @*/
public void dft() {
  /*...*/
}
```

Listing 4.8: Method-contract of dft

The loop inside `dft` is specified with loop-invariants instead of a loop-contract, which was used in the recursive form (see listing 3.6). This is because during the verification-process of recursive DFT, the proof of `dft`'s loop-contract turned out not to be as large as expected. Hence, the loop does not need to be verified in an external proof. That loop-contract's requirements match the loop-invariants used in listing 4.9. Additionally the loop-invariants keep track that the `stackPtr` must equal `0`, which means that before and after each iteration, the stack should be empty. Like the method-contract, the loop may additionally assign `before`, `diff`, `stackPtr` and `stack`'s entries.

```
/*@ loop_invariant 0 <= k && k <= G.size;
  @ loop_invariant \invariant_for(this);
  @ loop_invariant (\forall int i; 0 <= i && i < k; ord[i] != -1);
  @ loop_invariant cnt >= k && stackPtr == 0;
  @ decreases G.size - k;
  @ assignable unvisited(), cnt, k, before, diff, stack[*], stackPtr;
  @*/
for(int k = 0; k < /*G.size*/; k++) {
  if (ord[k] == -1){searchC(k);}
}
```

Listing 4.9: Loop-contract of dft

Since the `searchC`-method contains two loops, each having a contract, which specifies its behavior, we can brake down `searchC`'s specification into three parts: a method-contract and two block-contracts, one for each loop.

```
/*@ public normal_behavior
  @  requires cnt < G.size;
  @  requires 0 <= v && v < G.size;
  @  requires ord[v] == -1;
  @  requires stackPtr == 0;
  @
  @  ensures cnt > \old(cnt);
  @  ensures ord[v] == \old(cnt);
  @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @  ensures stackPtr == 0;
  @ assignable unvisited(), cnt, stack[*], stackPtr, before, diff;
  @*/
public void searchC(int v) {
  stack[stackPtr++] = v;

  while(stackPtr != 0) {
    /*...*/
  }

}
```

Listing 4.10: Method-contract of searchC

The method-contract of `searchC` in listing 4.10 requires that the given `int v` is a valid vertex ($v \in [0, G.size)$) and that `v` is not visited yet. This also means that `cnt` must be less than the number of vertices in `G`. Also the stack must be empty, so it requires that `stackPtr` is `0`. Before the `while`-loop starts, `v` is pushed onto the stack, which means that it is the next vertex to be visited. This is why the method-contract can ensure that `v` is visited afterwards and that `cnt` is increased by at least one. Also it ensures that `ord` contains a permutation of all newly visited vertices.

The following `while`-loop in listing 4.11 takes vertices off the stack and marks the unvisited ones as visited. So, in the first iteration, the stack only consists of vertex `v` at index 0. In the second iteration, `v` is already marked as visited and `cnt` has been increased by one. This is why there is a case-distinction in the loop-contract's second requirement. The left side of the `or` contains the requirements for the first iteration, the right side those for all later iterations. But in both cases the contract can ensure that after the execution of the loop's body, `v` is visited and `cnt` has been increased at least once. Since the loop terminates when `stackPtr` equals `0`, the loop-contract can also assure that the stack is empty after the last iteration. Also the loop-contract maintains the class-invariants.

In order to show that the loop terminates, the contract contains a `decreases`-clause. In each iteration a vertex is popped off the stack, which is either visited or unvisited. In case it has already been visited, the iteration is over, hence only the `stackPtr` is decreased by one. In the other case, `cnt` is increased by one and the `stackPtr` is increased by at most `G.size`. So progress towards termination is indicated by `cnt` increasing or `stackPtr` decreasing when `cnt`

remains untouched. So the sum of `-cnt` and `stackPtr` would result in a decreasing expression, if `stackPtr` would fall monotonically.But since it alternates, the change of `cnt` must be weighed with the maximum size that `stackPtr` can increase, which is `G.size`. That results into `-(cnt * G.size) + stackPtr`.

```
public void searchC(int v) {
  stack[stackPtr++] = v;

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires (stack[0] == v && ord[v] == -1 && stackPtr == 1 && cnt == \old(cnt))
            || (cnt > \old(cnt) && ord[v] == \old(cnt));
    @  ...
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @  ensures \old(cnt) < cnt;
    @  ensures ord[v] == \old(cnt);
    @  ensures stackPtr == 0;
    @  ensures \invariant_for(this);
    @ decreases (G.size * G.size) - (cnt * G.size) + stackPtr;
    @ assignable stackPtr, stack[*], cnt, unvisited(), diff, before;
    @*/
  {
    while(stackPtr != 0) {
      int u = stack[--stackPtr];
      stack[stackPtr] = -1;

      if (ord[u] == -1) {
        ord[u] = cnt++;

        /*List*/ adj = /*G.adjacent(v)*/;

        /*@ set diff = 0;
          @ set before = stackPtr;
          @*/

        for(/*int t : adj*/) {
          /*...*/
        }
      }
    }

  }
}
```

Listing 4.11: Loop-contract of searchC

As the `decreases`-expression must not fall below `0`, `G.size * G.size` is added to the expression, since `-(cnt * G.size)` will not drop below `-(G.size * G.size)`. Together with `stackPtr`

being larger than `0`, `(G.size * G.size) - (cnt * G.size) + stackPtr` falls monotonically and wont drop below `0`.

Finally, to be able to prove the termination of the `while`-loop, the next contract must assure that the second loop increases `stackPtr` by at most `G.size`. Therefore, the `ghost`-field `before` stores the `stackPtr`'s value before entering the next loop. The `ghost`-field `diff` is set to `0`. It will keep track of how much `stackPtr` changes during the loop.

The inner `for`-loop of `searchC` is specified with loop-invariants (see listing 4.13). These are additionally surrounded by a block-contract, which allows the inner loop to be verified in a separate, external proof. This block-contract (see listing 4.12) basically requires and ensures the invariants of the `for`-loop. Still it adds three requirements: `diff = 0` and `stackPtr = before` which can be met, because `diff` and `before` were set just before the block. The third one requires `stackPtr` to be less or equal to `cnt * G.size`, which differs from the class's invariant, which states: `stackPtr` $\leq$ `(cnt + 1) * G.size`. The stricter requirement can still be met, since in the `while`-loop (see listing 4.11) `cnt` has been increased by one, but no new vertices have been pushed on the stack yet, so `stackPtr` has also not been increased. With this precondition and the limit of `stackPtr`, changing no more than `G.size`, the just mentioned class-invariant can be shown to be correct after the loop.

```
/*List*/ adj = /*G.adjacent(u)*/;

/*@ set diff = 0;
  @ set before = stackPtr;
  @*/

/*@  requires diff == 0
  @   requires stackPtr <= (cnt * G.size);
  @   requires stackPtr == before;
  @    ...
  @*/
{
  for(/*int t : adj*/) {
    /*...*/
  }
}
```

Listing 4.12: Block-contract of searchC

Listing 4.13 shows the inner `for`-loop of `searchC` and its invariants. Inside the loop's body, there is a `set`-statement, which, in case an unvisited adjacent vertex is pushed onto the stack, increases the `diff-ghost`-field by one. In this way `diff` counts by how much `stackPtr` changes. This is reflected in the third loop-invariant, which states that `stackPtr` equals its value before the block (`before`) plus the value it has already changed (`diff`). Also the value of `diff` can be computed by counting the adjacent vertices to `u` in `adj` that are unvisited and whose index in `adj` is in $[0, t)$ (which means that the loop has already looked at this element in `adj`). Since `diff` only counts unvisited vertices, its value cannot exceed the number of vertices in `G` minus

the number of visited vertices `cnt`. All these invariants are later used to prove that `stackPtr` changes by at most `G.size` and that it does not go out of `stack`'s bounds.

```
/*@ loop_invariant \invariant_for(this);
  @ ...
  @ loop_invariant diff <= ord.length - cnt;
  @ loop_invariant stackPtr == before + diff;
  @ loop_invariant //Number of unvisited vertices in adj before t// == diff;
  @
  @ decreases adj.length - t;
  @ assignable stack[stackPtr..stack.length], stackPtr, t, diff;
  @*/
for(/*int t : adj*/) {
  if (ord[t] == -1) {
    /*@ set diff = diff + 1;
      @*/
    stack[stackPtr++] = t;
  }
}
```

Listing 4.13: Loop-invariants of searchC's inner for-loop

4.3 Verification-Process and Interactive-Steps

During the verification of non-recursive DFT the taclets used for verifying the recursive form can be reused (see section 3.3). This is because the interactive steps described in section 3.3 need to be applied in the non-recursive proofs as well. Since the `dft`-methods do not vary, the first two interactions (see section 3.3) appear at a similar positions in the `dft`-contract proof. Due to the fact that the loop in `dft` is here specified with loop-invariants instead of a loop-contract, which was used in the recursive form, those interactions are not located in an external proof but in the `Body-Preserves-Invariant` branch of the loop. The third one of the mentioned interactions reappears in the proof of `searchC`'s loop-contract, when establishing the class-invariants in the following block-contract's precondition.

Apart from those reappearing interactions, some new ones are required in order to close the proofs. For example, the `decreases`-clause of `searchC`'s loop-contract: `(G.size * G.size) - (cnt * G.size) + stackPtr` is more complex, than the recursive one: `G.size - cnt`. Also proving the invariant `stackPtr <= (cnt + 1) * G.size` to remain correct requires some attention. Depending on the used data structure, the attention they need varies. Some of these goals can not be closed by KeY with its `Basic Arithmetic Treatment`, but selecting an advanced `Arithmetic Treatment` (`DevOps` or `Model Search`) in the `Proof Search Strategies` suffices to close them. Others, which can not be closed with advanced `Arithmetic Treatment`, are closed by the SMT-solver Z3 [MB13] after a few interactive steps. Still others can not be closed by Z3 and need to be closed completely interactively.

Furthermore, there are two new points, where the application of the defined taclets (see section 3.3) is required. Both appear while proving the block-contract of `searchC`'s inner `for`-loop (see listing 4.12), in the loop's `Body Preserves Invariant`-branch in the case where

one of the vertices in `adj` is pushed onto the stack. Since the implementation of the loop slightly varies for each data structure, we will look at the proofs for the adjacency matrix version. Still, the general strategy behind the proofs remains the same for all data structures.

The first goal is to prove that $1 + diff \leq ord.length - cnt$. The `ghost`-field *diff* is used to count by how much *stackPtr* was already increased during the `for`-loop. Also *diff* equals the number of unvisited, adjacent vertices of the vertex u that have already been treated by the loop. Furthermore, `cnt` counts the visited vertices in `G`. The mentioned goal is one of `searchC`'s `for`-loop's postconditions. It states that the number of visited vertices together with the number of unvisited, adjacent vertices can not be larger than the total number of vertices in the graph (*ord.length*). This goal can be reduced to showing that $diff = ord.length - cnt$ is wrong.

$$\begin{aligned} ord[t] &= -1, \\ cnt &= \sum_{i=0}^{<ord.length} \delta(ord[i] \neq -1), \\ diff &= \sum_{i=0}^{<t} \delta(ord[i] = -1 \land adj[i]), \\ diff &= ord.length - cnt \implies 1 + diff \leq ord.length - cnt \end{aligned} \tag{4.1}$$

Equation (4.1) shows that goal. The sequent's left side contains the goal's assumptions and its right side contains the postcondition we want to prove. We find four terms on the sequent's left side, which are important in the following. For one $ord[t]$ equals -1, where t is the current value of the loops counting-variable. Also there are the bounded sum expressions of *cnt* and *diff*. Finally, there is the fourth equation, we want to prove wrong.

Here *adj* is the *boolean*-array $G[u]$ and $G[u][i]$ contains the information, whether i is adjacent to u (see section 4.4). The idea behind the proof is that there are *ord.length* vertices in the graph, of which *cnt* are visited. Furthermore, $ord.length - cnt$ vertices are unvisited and adjacent to u. As there cannot be any more vertices and the unvisited ones are summed up by *diff*'s sum in (4.1), all unvisited vertices must be in $[0, t)$. This leads to a contradiction, since the vertex t is also unvisited. At first, we can show with *int-induction* that weakening the guard of *diff*'s sum in (4.1) can only result in a greater value:

$$(4.1) \implies \sum_{i=0}^{<t} \delta(ord[i] = -1) \geq \sum_{i=0}^{<t} \delta(ord[i] = -1 \land adj[i]) = diff \tag{4.2}$$

Again with *int-induction*, we can prove two more properties. On the one hand, increasing the fist sum's upper bound in (4.2) to *ord.length* can only increase its value:

$$(4.1) \implies \sum_{i=0}^{<ord.length} \delta(ord[i] = -1) \geq \sum_{i=0}^{<t} \delta(ord[i] = -1) \overset{(4.2)}{\geq} diff \tag{4.3}$$

On the other hand, adding two bounded sums, where each guard is the negative of the other, equals the number of the summed elements:

$$\sum_{i=0}^{<ord.length} \delta(ord[i] = -1) + \underbrace{\sum_{i=0}^{<ord.length} \delta(ord[i] \neq -1)}_{\overset{(4.1)}{=} cnt} = ord.length \tag{4.4}$$

In (4.4) the second sum equals cnt, so the former sum equals $ord.length - cnt$. With that, the following can be deduced:

$$\begin{aligned}(4.1) \implies ord.length - cnt &\overset{(4.4)}{=} \sum_{i=0}^{<ord.length} \delta(ord[i] = -1) \overset{(4.3)}{\geq} \\ \sum_{i=0}^{<t} \delta(ord[i] = -1) &\overset{(4.2)}{\geq} diff \overset{(4.1)}{=} ord.length - cnt\end{aligned} \tag{4.5}$$

As $ord.length - cnt$ is both the minimum and the maximum value in the chain of inequations of (4.5), the sums in between must equal $ord.length - cnt$ as well:

$$\sum_{i=0}^{<t} \delta(ord[i] = -1) \overset{(4.5)}{=} ord.length - cnt \tag{4.6}$$

.

$$\sum_{i=0}^{<ord.length} \delta(ord[i] = -1) \overset{(4.5)}{=} ord.length - cnt \tag{4.7}$$

Now we can split the sum in (4.7) at t into three parts:

$$\underbrace{\sum_{i=0}^{<t} \delta(ord[i] = -1)}_{\overset{(4.6)}{=} ord.length - cnt} + \underbrace{\delta(ord[t] = -1)}_{\overset{(4.1)}{=} 1} + \underbrace{\sum_{i=t+1}^{<ord.length} \delta(ord[i] = -1)}_{\overset{(3.3)}{\geq} 0} = ord.length - cnt \tag{4.8}$$

As we just established in (4.6), the first sum in (4.8) equals $ord.length - cnt$, so the other two summands must equal 0. But the *bsum_num_of_bounds*-taclet (3.3) tells us that the second sum is at least 0. Also $\delta(ord[t] = -1)$ resolves to 1. This resolves to $false$ on the sequent's left side, which closes this branch.

The second goal is to show that $stackPtr < G.length + cnt * G.length$ is true. Since *stackPtr* equals $before + diff + 1$ and the block's precondition provides $before \leq cnt * G.length$, the goal reduces to $1 + diff < G.length$. Similar to the previous goal, *diff* counts the unvisited, adjacent vertices of the vertex u that have already been treated by the loop. Also `cnt` counts the visited

vertices in G. The length of G denotes the number of vertices in the graph. Here it suffices to prove that $1 + diff = G.length$ is wrong, the other case can then be closed by KeY.

$$\begin{aligned} ord[t] &= -1, \\ adj[t] &= TRUE, \\ ord[v] &\neq -1, \\ diff = \sum_{i=0}^{<t} \delta(ord[i] = -1 \land adj[i]), & \\ 1 + diff = G.length \implies 1 + diff < G.length & \end{aligned} \tag{4.9}$$

In (4.9), we see that goal consisting of the assumptions on the sequent's left side and the property we want to prove on its right side. On the sequent's left side, we find again *diff*'s bounded-sum expression, as well as $ord[t] = -1$, $adj[t] = TRUE$ and $ord[v] \neq -1$. The index v represents that vertex that was visited at first in the current call of `searchC` and t is the current value of the `for`-loops counting variable. Finally, the equality, we will show to be wrong can be found there. The idea of this proof is to show that if $1 + diff = G.length$ were true, then all vertices would be unvisited, which is wrong, since v has already been visited. At first we can increase the upper bound of *diff*'s sum in (4.9) by one, so it includes t. Since the sums guard is true for t, 1 is added to the left side of the equation:

$$(4.9) \implies diff + 1 = \sum_{i=0}^{<t+1} \delta(ord[i] = -1 \land adj[i]) \tag{4.10}$$

The equality of $diff + 1$ to $G.length$ (see (4.9)) can be applied on its appearance in (4.10). When increasing the sums upper bound to $G.length$ in (4.10), the resulting value can only increase, which can be proven with *int-induction*:

$$(4.9) \implies \underbrace{\sum_{i=0}^{<G.length} \delta(ord[i] = -1 \land adj[i])}_{\overset{(3.3)}{\leq} G.length} \geq G.length \tag{4.11}$$

The *bsum_num_of_bounds*-taclet (3.3) provides that the sum in (4.11) is at most $G.length$, so the inequality ($\geq$) is actually an equality ($=$):

$$(4.11) \implies \sum_{i=0}^{<G.length} \delta(ord[i] = -1 \land adj[i]) \overset{(4.11)}{=} G.length - 0 \tag{4.12}$$

Now the taclet *bsum_num_of_is_max* (3.4) can be applied on (4.12), which states that the following is true:

$$(4.12) \overset{(3.4)}{\implies} \forall i : (0 \leq i < G.length \rightarrow ord[i] = -1 \land adj[i]) \tag{4.13}$$

When instantiating i in (4.13) with v, $ord[v] = -1$ is produced, which is a contradiction to the assumption $ord[v] \neq -1$ in (4.9).

4.4 Adjacency Matrix

The adjacency matrix is implemented as two-dimensional `boolean` array, as it is in the recursive form (see section 3.4). Hence, there are also the same class-invariants, which state that `G` is a square matrix and that it contains at least one vertex (shown in listing 4.14).

```
/*@ public invariant (\forall int j; 0 <= j && j < G.length;
                      G[j] != null && G[j].length == G.length);
  @ public invariant G.length > 0;
  @*/
boolean[][] G;
```

Listing 4.14: Invariants of the adjacency matrix

Also the retrieval of `u`'s adjacent vertices in `searchC`'s `while`-loop is similar to the recursive from. Here (see listing 4.15), `adj` is set to `G[u]` and contains the information for each vertex, whether it is adjacent to `u`. This is why the `if`-guard in the `for`-loop must check whether `t` is adjacent to `u`, additionally to checking whether it is unvisited, before pushing it onto the stack. Also the specification of the `diff`-variable depends on the data structure. Here `diff` equals the number of fields `adj[i]` where `i` $\in$ `[0,t)` and both `adj[i]` and `ord[i] = -1` are true.

```
boolean[] adj = G[u];

/*@  requires \invariant_for(this);
  @  requires adj == G[u];
  @  requires 0 <= u && u < G.length;
  @*/
{
  /*@ loop_invariant (\num_of int i; 0 <= i && i < t; ord[i] == -1 && adj[i]) == diff;
    @ ...
    @*/
  for(int t = 0; t < adj.length; t++) {
    if (adj[t] && ord[t] == -1) {
      /*...*/
    }
  }
}
```

Listing 4.15: Retrieval of adjacent vertices using the adjacency matrix

Except those interactive steps that all data structures required with non-recursive DFT (see section 4.3), the verification with adjacency matrix did not require any additional interactions.

Of all four data-structures, the adjacency matrix is the only one with which non-recursive DFT is proven.

4.5 Adjacency Array

The implementation and specification of the adjacency array representation for non-recursive DFT also equal those for recursive DFT (cf. section 3.5 and listing 4.16). The `adjArr`-array is a concatenation of all the vertices adjacency lists and the interval `[G[u],G[u+1])` describes the range in `adjArr`, where `u`'s adjacent vertices are stored.

```
/*@ public invariant (\forall int j; 0 <= j && j < len;
                        0 <= G[j] && G[j] < adjArr.length && (G[j] <= G[j+1]));
  @ public invariant len > 0;
  @ public invariant G != adjArr && adjArr != ord && ord != G
                                  && G != stack && adjArr != stack;
  @ public invariant 0 <= G[len] && G[len] < adjArr.length;
  @*/
int[] G;

/*@ public invariant (\forall int i; 0 <= i && i < adjArr.length;
                      0 <= adjArr[i] && adjArr[i] < len);
  @*/
int[] adjArr;
```

Listing 4.16: Invariants of the adjacency array

This is why `searchC`'s inner `for`-loop only needs to iterate that range instead of the whole `adjArr`-array. Also the `\num_of`-expression, which equals `diff`, must be adapted to the data structure. As shown in listing 4.17, `G[u]` is the lower bound of the iterated range, which also is the lower bound of the sum. The expression counts those entries `i`, where the adjacent vertex at `i` (`adjArr[i]`) is unvisited.

```
/*@ requires 0 <= u && u < len;
  @ ...
  @*/
{
  /*@ loop_invariant G[u] <= t && t <= G[u+1];
    @ loop_invariant 0 <= u && u < len;
    @ ...
    @ loop_invariant (\num_of int i; G[u] <= i && i < t; ord[adjArr[i]] == -1) == diff;
    @*/
  for(int t = G[u]; t < G[u+1]; t++) {
    if (ord[adjArr[t]] == -1) {
      /*...*/
    }
  }
}
```

Listing 4.17: Retrieval of adjacent vertices using the adjacency array data structure

The verification-attempt non-recursive DFT with the adjacency array data structure required similar interactions as the recursive one (see section 3.5). Additionally, proving that `adjArr[t]` is in `[0,len)` in `searchC`'s inner `for`-loop required interactive help. The first invariant in

listing 4.16 can be instantiated with u and together with the loop-invariant $G[u] \leq t \leq G[u+1]$ this implies $0 \leq t < adjArr.length$. As t is in its range, the universal quantifier of the last invariant in listing 4.16 can be instantiated with t, which resolves in the desired $0 \leq adjArr[t] < len$.

The proof of non-recursive DFT with adjacency array was not closed and there are 7 open goals left. When using the automatic proof-preparation in KeY on the initial proof-obligation, a proof-tree with 327 open goals is created. Thus, after the proof-attempt, there are about 2% of the initial goals left open.

4.6 Linked Data Structure

Like with the recursive implementation (see section 3.6), the linked data structure for non-recursive DFT uses `Vertex`-objects. Each `Vertex` holds an `int`-id `val` and a `Vertex`-array of its adjacent vertices. The non-recursive-DFT class holds an array `G`, which contains all of the graphs vertices (see listing 4.18).

```
/*@ public invariant (\forall int j; 0 <= j && j < G.length;
                       G[j] != null && G[j].adj.length <= G.length);
  @ public invariant G.length > 0;
  @ public invariant (\forall int i; 0 <= i && i < G.length;
                      (\forall int j; 0 <= j && j < G[i].adj.length;
                        0 <= G[i].adj[j].val && G[i].adj[j].val < G.length
                        && G[i].adj[j] == G[G[i].adj[j].val]));
  @ public invariant (\forall int i; 0 <= i && i < G.length; G[i].val == i);
  @ public invariant (\forall int i; 0 <= i && i < G.length;
                      (\forall int j; 0 <= j && j < G.length;
                        (i!=j) ==> (G[i] != G[j]) ));
  @ public invariant (\forall int i; 0 <= i && i < G.length; \invariant_for(G[i]));
  @*/
Vertex[] G;
```

Listing 4.18: Invariants of the linked data structure

To push `u`'s unvisited adjacent vertices onto the stack, `searchC`'s inner loop iterates the `Vertex`-array `G[u].adj`. To check, whether an adjacent vertex `adj[i]` is unvisited, its index in the `ord`-array must be obtained, which is stored in its `val`-field: `adj[i].val`. Looking up, if `ord` is `-1` at that index, decides whether the vertex is pushed onto the stack. So `ord[adj[i]] = -1` forms the guard inside the `for`-loop and also in the `\num_of`-expression.

Like the recursive version (see section 3.5), the verification of non-recursive DFT with the linked data structure required interactive help with `assignable`-clauses and for proving that all `Vertex`'s invariants remain true. Additionally the `decreases`-clause of `searchC`'s loop-contract and `stackPtr`'s invariant `stackPtr <= (cnt + 1) * G.size` required interactive steps, as described in section 4.3.

```
Vertex[] adj = G[u].adj;

/*@  requires adj == G[u].adj;
  @  requires 0 <= u && u < G.length;
  @  ...
  @*/
{
  /*@ loop_invariant adj == G[u].adj;
    @ loop_invariant 0 <= u && u < G.length;
    @ ...
    @ loop_invariant (\num_of int i; 0 <= i && i < t; ord[adj[i].val] == -1) == diff;
    @*/
  for(int t = 0; t < adj.length; t++) {
    if (ord[adj[t].val] == -1) {
      /*...*/
    }
  }
}
```

Listing 4.19: Retrieval of adjacent vertices using the linked data structure

Applying the automatic proof-preparation in KeY on the initial proof-obligation results in a proof-tree with 339 open goals. After the verification process for non-recursive DFT with the linked data structure there are 25 open goals left. This makes about 6% of the initial open goals.

4.7 Adjacency Lists

The adjacency lists implementation for non-recursive and recursive DFT (see section 3.7) are identical. It uses a `List` interface, which allows storing `int` values and provides a `size()`-method for obtaining the `List`'s size as well as a `get(int i)`-method, which returns the `i`-th element of the `List`. Thus the graph `G` is an array of `List`-objects, so that `G[i]` is the adjacency list of the vertex `i`. Similar to the recursive version, the locations of a lists elements should not alias with the `cnt`-field and `ord`'s entries. Since the non-recursive version uses a stack, also the `stackPtr` and `stack`'s entries can be manipulated and might alias with the elements of the lists, thus they are included in the last invariant.

The inner `for`-loop in `searchC` iterates `G[u]` to check u's adjacent vertices. If the `t`-th adjacent vertex `adj.get(t)` is unvisited, it is pushed onto the stack. This is why `ord[adj.get(t)]` is the guard of the `if`-statement and the `\num_of`-expression in listing 4.21.

```
/*@ public invariant (\forall int j; 0 <= j && j < G.length;
                      G[j] != null && G[j].size() <= G.length);
  @ public invariant G.length > 0;
  @ public invariant (\forall int i; 0 <= i && i < G.length;
                      (\forall int j; 0 <= j && j < G[i].size();
                       0 <= G[i].get(j) && G[i].get(j) < G.length));
  @ public invariant (\forall int i; 0 <= i && i < G.length; \invariant_for(G[i]));
  @ public invariant (\forall int i; 0 <= i && i < G.length;
                      \disjoint(G[i].footprint,\singleton(cnt))
                    && \disjoint(G[i].footprint, \singleton(stackPtr))
                    && (\forall int j; 0 <= j && j < ord.length;
                    \disjoint(G[i].footprint,\singleton(ord[j])))
                    && (\forall int l; 0 <= l && l < stack.length;
                        \disjoint(G[i].footprint,\singleton(stack[l]))));
  @*/
List[] G;
```

Listing 4.20: Invariants of adjacency lists

```
List adj = G[u];

/*@ requires 0 <= u && u < G.length;
  @ requires adj == G[u];
  @ ...
  @*/
{
  /*@ loop_invariant 0 <= u && u < G.length && adj == G[u];
    @ ...
    @ loop_invariant (\num_of int i; 0 <= i && i < t; ord[adj.get(i)] == -1) == diff;
    @*/
  for(int t = 0; t < adj.size(); t++) {
    if (ord[adj.get(t)] == -1) {
      /*...*/
    }
  }
}
```

Listing 4.21: Retrieval of adjacent vertices using adjacency lists

The proof-attempt of non-recursive DFT with adjacency lists requires similar interactions as the recursive form (see section 4.7). Additionally those interactions described in section 4.3 were required, including the ones, where `Basic Arithmetic Treatment` can not close the branch.

The proof for non-recursive DFT with adjacency lists leaves 64 goals open. The automatic proof-preparation in KeY creates a proof-tree with 422 open goals, when used on the initial proof-obligation. In this way there are about 15% of the initial goals left open, after the proof-attempt.

5 Depth-First Property

In chapters 3 and 4 we consider the DFT-algorithm to be correct if the whole graph has been traversed after it terminates, meaning that all vertices have been visited once. This specification was sufficient, to compare the suitability of the data structures in the verification-process with KeY. In the following, we will define a stricter depth-first property, which describes that the vertices in a graph traversal are actually visited in a depth-first order. Additionally we will show, how this property can be translated to JML and verified with KeY.

In the following, we will write $\exists i \in [0..n] : phi(i)$ to denote that there exists an i, such that i is in the range of $[0, n]$ and $phi(i)$ is true. Similar, we write $\forall i \in [0..n] : phi(i)$, to describe that for all i, if i is in the range of $[0, n]$, then $phi(i)$ is true.

First of all, we need to define, what a graph traversal is, in order to describe which graph traversals are depth-first traversals.

Definition 9 (Graph traversal, partial graph traversal) *A graph traversal of a graph $G = (V, E)$, is a sequence of vertices $t = (v_0, \ldots, v_m)$, where every vertex $v \in V$ appears in t exactly once.*

A partial graph traversal of a graph G is a graph traversal of a graph G', such that G' is a subgraph of G.

By definition 9, a graph traversal is an arbitrary permutation of a graphs vertices. This order could stem from an algorithm, performing breadth-first search, depth-first search or just selecting random vertices.

When a yet unvisited vertex v_m is visited, it can be added to the end of graph traversal $t = (v_0, \ldots, v_{m-1})$. In this way, the index i of a vertex $v_i = t_i$ (the i-th vertex in t) denotes when the vertex has been visited.

Definition 10 *Let $t = (v_0, \ldots, v_m)$ be a graph traversal and let i and j be indices, such that $0 \leq i < j \leq m$, then the vertex $v_i = t_i$ is said to be visited before $v_j = t_j$.*

With definition 10, the subsequence $(v_0, \ldots, v_{i-1}) \subset t$ consists of vertices that are visited before v_i and the subsequence $(v_{i+1}, \ldots, v_m) \subset t$ contains those that are visited after v_i.

Definition 11 (Parent, child) *A vertex u is a parent of a vertex v in a graph $G = (V, E)$ iff there is an edge from the former to the latter. The vertex v is then called child of u. The predicate $parent(u, v)$, expressing that u is parent of v, is defined as follows:*

$$parent(u, v) \Leftrightarrow (u, v) \in E$$

If a vertex v_n is visited during depth-first search it must have a parent v_p, which is visited before v_n. The only vertex that does not have visited a parent is the root v_0, which is the first vertex to be visited. Also, v_n's parent v_p may have other children besides v_n. If such a child $v_c \neq v_n$ is visited before v_n, it must be finished by the DFS-procedure, before v_n is visited. If the vertex v_c is finished before v_n that means it is finished on the partial graph traversal $t = (v_0, \ldots, v_{n-1})$. We can say that a vertex is finished in a graph traversal, if itself and all its children are visited in that graph traversal.

Definition 12 (Finished vertex) *A vertex v_i in a partial graph traversal $t = (v_0, \ldots, v_m)$ of length m on a graph $G = (V, E)$ is finished iff v_i and all children of v_i appear in t, which means they have been visited. The predicate finished(i, t) describes this property and is defined as follows:*

$$finished(i, t) \Leftrightarrow 0 \leq i \leq m \wedge \forall y \in V : (parent(t_i, y) \implies \exists c \in [0..m] : t_c = y)$$

We can now define, when visiting such a vertex v_n is valid with respect to depth-first search.

Definition 13 (Valid DFS vertex) *The last vertex v_m in a partial graph traversal $t = (v_0, \ldots, v_m)$ of length m on a graph $G = (V, E)$ is a valid DFS vertex iff either v_m is the root v_0 or there is a parent-vertex v_p of v_m in t, such that all vertices in $(v_{p+1}, \ldots, v_{m-1})$ are finished in $t' = (v_0, \ldots, v_{m-1})$. The predicate validDFSvertex(t) states whether the last vertex v_m in t is a valid DFS vertex and is defined as follows:*

$$validDFSvertex(t) \Leftrightarrow (m = 0) \vee \exists p \in [0..m) : (parent(t_p, t_m) \wedge \forall c \in (p..m) : finished(c, t'))$$

We are using DFT instead of DFS in chapters 3 and 4, which can execute DFS multiple times. When DFS is started at the vertex v_0 and all vertices that are reachable from v_0 have been visited by DFS, DFT must check whether the whole graph has been visited. In case it is not, DFT has to restart DFS at a vertex v_m that has not been visited yet. Thus, a vertex is either a vertex where DFS is (re)started or it is visited during DFS but is not the root. Such a restart vertex can only be visited if the previous DFS has finished.

Definition 14 (Valid DFT vertex) *The last vertex v_m in a partial graph traversal $t = (v_0, \ldots, v_m)$ of length m on a graph $G = (V, E)$ is a valid DFT vertex iff either it is a valid DFS vertex (definition 13) or all vertices visited before v_m are finished in $t' = (v_0, \ldots, v_{m-1})$. The predicate validDFTvertex(t) states whether the last vertex v_m in t is a valid DFT vertex and is defined as follows:*

$$validDFTvertex(t) \Leftrightarrow validDFSvertex(t) \vee \forall c \in [0..m) : finished(c, t')$$

$$\Leftrightarrow \exists p \in [-1..m) : ((parent(t_p, t_m) \vee p = -1) \wedge \forall c \in (p..m) : finished(c, t'))$$

Definition 15 (Depth-first graph traversal) *A graph traversal $t = (v_0, \ldots, v_m)$ on a graph $G = (V, E)$ is a depth-first graph traversal iff for each partial graph traversal $t^n = (v_0, \ldots, v_n), (0 \leq n \leq m)$ the vertex v_n is a valid DFT vertex. For $t^n = (v_0, \ldots, v_n)$, $t'^n = (v_0, \ldots, v_{n-1})$.*

$$DFgraphTraversal(t) \Leftrightarrow \forall n \in [0..m] : validDFTvertex(t^n)$$

Now that we have a definition of the depth-first property (see definition 15), we can translate it to JML, in order to specify a DFT-implementation. As working with the recursive version of DFT (see chapter 3) using an adjacency matrix (see section 3.4) was easy to work with (see section 6.4), we will specify the depth-first property for that implementation.

First of all, a `int ghost`-array `t` can be used as sequence of vertices, representing the graph traversal. The entries of `t` contain vertices `t[n] = v`, such that `v` is the n-th visited vertex in the graph traversal. If not all vertices are visited yet, there is one most recently visited vertex `w = t[m]` and all entries `t[m+1..t.length]` are `-1`. The implementation contains an `int`-field `cnt`, which counts the visited vertices, thus the most recently visited vertex is `t[cnt-1]`. Listing 5.1 shows, how in the `searchC`-method, the given vertex `v` is added to the graph traversal `t`, using a `set`-statement. After that, `v` is marked as visited in the `ord`-array before the `cnt`-field is increased by one, since another vertex has been visited. The `ord`-array is inverse to the `t`-array, as `ord[v] = n` says that `v` is the n-th visited vertex, which is also `t[n]`.

```
public void searchC(int v) {
  /*@ set t[cnt] = v;
    @*/
  ord[v] = cnt++;
  /*...*/
}
```

Listing 5.1: Adding a visited vertex `v` to the graph traversal `t`

With this representation of a graph traversal, we can translate the predicates $parent(t_v, t_w)$ and $finished(w, t'^n)$. A vertex v_w is the w-th visited vertex and is stored at `t[w]`. Since the implementation uses the adjacency matrix data structure, the $parent(t_v, t_w)$-predicate can be translated to a lookup in that matrix: `G[`t_v`][`t_w`] = G[t[v]][t[w]]`. A vertex v_w is marked as visited, if `ord[`v_w`]` is not -1. Definition 12 says that a vertex v_w is finished in $t = (v_0, \ldots, v_m)$ if itself and all its children appear in t. Here we use the *finished*-predicate to say that v_w is finished in the partial graph traversal $t'^n = (v_0, \ldots, v_{n-1})$. From the definition we get that v_w and its children must appear in `t` with indices in $[0, n-1] = [0, n)$. Listing 5.2 shows that $finished(w, t'^n)$ can be translated to JML, using a universal quantifier over all vertices `y` in `G`. The body of `\forall` states that, if v_w is parent of `y`, vertex `y` must be visited before v_n, meaning that `ord[y]` must be in `[0,n)`. The definition of the *finished*-predicate contains an existential quantifier, which states that for a child y of t_i there exists an index c in t, such that $t_c = y$. In the JML-specification, this quantifier can be replaced by `ord[y]`, since the `ord`-array is inverse to the `t`-array and contains each vertex's index in `t`.

```
  /* w < n & (\forall int y; 0 <= y && y < G.length; G[t[w]][y] ==> (0 <= ord[y] &&
      ord[y] < n)))
  */
```

Listing 5.2: Translation of the finished-predicate to JML

When using the translation in listing 5.2 to translate $finished(t'_n[w], t'_n)$ in listing 5.3, we can leave out the condition `w < n`, as it is nested in a universal quantifier over those `w` in `(p,n)`. Hence `w < n` is always true. With this, we can give a class-invariant (see listing 5.3), which

states that at every time, the current partial graph traversal `t[0..cnt]` fulfills the depth-first condition in definition 15.

```
/*@ public invariant (\forall int v; 0 <= v && v < cnt;
                        (\exists int p; -1 <= p && p < v;
                            (G[t[p]][t[v]] || p == -1)
                             && (\forall int w; p < w && w < v;
                                   (\forall int y; 0 <= y && y < G.length;
                                      G[t[w]][y] ==> (0 <= ord[y] && ord[y] < v)))
                        )
                       );
  @ ghost int[] t;
  @*/
```

Listing 5.3: Specification of the depth-first property in JML

Here the predicates were translated for the adjacency matrix implementation. For other data structures, the translation would look similar, yet slightly different, as the adjacency of two vertices is represented differently. Additionally to the invariant in listing 5.3, a few other invariants are required. On the one hand, there are unassigned entries in the `t`-array with a default value of `-1`. As there are `cnt` visited vertices, one class invariant states that the range `t[0..cnt]` is assigned (not `-1`) and the range `t[cnt..t.length]` is unassigned and has only default entries. Furthermore, the class needs one invariant describing that `ord` is inverse to `t` and vice versa. Additionally to the class-invariants, some additional pre- and postconditions are added to the methods `dft` and `searchC` (see listing 3.2). The `searchC`-method takes a vertex `v`, from where it starts depth-first search. Hence it requires `v` to be a valid DFT vertex. As `searchC(v)` traverses all vertices, reachable from `v`, it ensures that `v` and all recursively visited vertices during that call are finished when `searchC(v)` returns. In the `dft`-method, `searchC` is called for the next vertex that was not visited during the previous call of `searchC`. This means that in every iteration, all vertices in the graph traversal `t` must be finished, before depth-first search is started anew with another call of `searchC`.

With these invariants, pre- and postconditions, the depth-first property was proven for the recursive implementation of DFT using an adjacency matrix.

6 Evaluation

In this chapter we will evaluate the specification- and verification processes for all four data structures. In section 6.1, the time that was spent on each proof will be evaluated and in section 6.2 we will evaluate the proof statistics provided by KeY. These evaluations provide data for both forms of DFT with all four data structures. Nevertheless, we have to keep in mind that the non-recursive form was only verified for the adjacency matrix version, while the other non-recursive proofs are not closed. Finally, in sections 6.3 and 6.4, both forms of DFT will be compared and the advantages and disadvantages for each data structure will be discussed. All the data that will be shown in the following does not show the optimal time needed to finish a proof or the minimum number of interactions, needed to close a proof, but rather the time and the number of interactions that I (resp. a common KeY user) needed to close these proofs. Thus the comparison of the data structures will also contain personal impressions from the working processes.

6.1 Evaluation of Time Effort

In order to estimate how much time was spent on each data structure and algorithm-form, all files in the working directory were committed into a git-repository. Additionally the *git-autocommit*-script was used (provided by [Hol17]). This script checks every minute, whether there are any changed files in the repository and commits them. In this way, the time between two commits can be assumed to be the time spent on the committed files. So after the specification- and verification-processes the commit-times could simply be read from the `git-log`, to compute the time efforts. To receive the changed files and time of each commits, the `git-log`-command can be used with the `-stat` option (which adds the changed files). In order to make the obtained data easier to process, the `-pretty=format`-option can be added, as listing 6.1 shows.

```
git log --stat --pretty=format:' "},{"date":"%ai","body":"%B","change":"%n ' > time.txt
```

Listing 6.1: Git-log-command for time evaluation

The `pretty`-option formats the output in a *json*-ish structure, so that each *json*-object contains a `date`-string (inserted by `% ai`), containing the date and time of the commit. Also each *json*-objects has a `body`-string (inserted by `% B`), containing the commit message, and a string `change` (inserted by `% n`), which contains the names of the commits changed files. Listing 6.2 shows an example of a commit as *json*-object.

```
{ "date":"2018-06-25 15:02:58 +0200",
  "body":"<git-autocommit>",
  "change":
     "simple_outer_searchC(int)_v12.proof | 17857 ++++++++++++++++++
     1 file changed, 17857 insertions(+)"
}
```

Listing 6.2: Example of a cajson-commit-object

The command in listing 6.1 writes the data into the file `time.txt`. Now all commits exist as comma-separated *json*-objects. By surrounding the generated *json*-objects with `{"commits":[...]}`, they become elements of the *json*-array `commits`.

Now the commits can be processed by a *processing*-script (see listing A.15), which parses the `date`-string and splits those commits that contain more than one file into several commits, each containing one file with the same `date`.

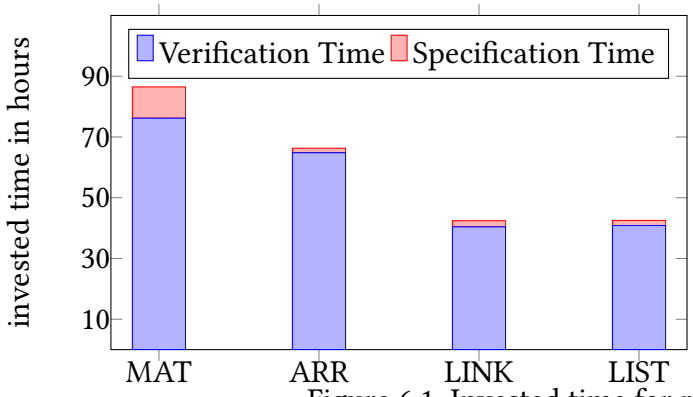
The script saves the processed data as *csv*-file, which can then be imported into *Excel* for further evaluation. Now we can compute the time difference of each commit to the previous one. Since the first commit of a day is committed several hours after the last one, we need to replace these times with more plausible values. Therefore we can calculate the average time per commit of previous and later commits of that same proof- or source-file. After that, each row contains the time that was spent on it. We can add three categories to each row, to divide the files in to different classes. The first category describes the data structure (DS) treated in the file (adjacency matrix (*MAT*), adjacency array (*ARR*), linked data structure (*LINK*) or adjacency lists (*LIST*)). The second one states, whether a file treated the recursive (*REC*) or non-recursive form (*NONREC*) of DFT. The last category distinguishes between specification (*SPEC*) and verification (*VER*). Table 6.1 shows some example data from the *Excel*-file with the classification of the commited files. The column 'Time spent' contains time values that consist of hours, minutes and seconds (h:mm:ss).

Table 6.1: Examplary commit-data with file classification

Time	File	DS	Form	Type	Time spent
15:50	GraphDFTalt_loop_contracts.java	MAT	REC	SPEC	0:01:06
15:52	GraphDFTalt_loop_contracts.java	MAT	REC	SPEC	0:01:04
16:17	searchC(int)_outer_v41.proof	MAT	REC	VER	0:25:45
16:29	GraphDFTalt_loop_contracts2.java	MAT	REC	SPEC	0:11:43
	...				
20:18	GraphDFT_link_nonrec.java	LINK	NONREC	SPEC	0:05:08
20:19	GraphDFT_link_nonrec_dft_v16.proof	LINK	NONREC	VER	0:01:03
20:21	GraphDFT_adjArr_nonrec.java	ARR	NONREC	SPEC	0:02:04
20:22	GraphDFT_list_nonrec.java	LIST	NONREC	SPEC	0:01:01
20:29	GraphDFT_link_nonrec_dft_v17.proof	LINK	NONREC	VER	0:07:12

With this classification, the time can be summed up for each combination of theses three categories: {*MAT*, *ARR*, *LINK*, *LIST*}×{*REC*, *NONREC*}×{*SPEC*, *VER*}. In this way, we receive the specification- and verification-time for each proof.

The table in figure 6.1 shows the time that was spent on the specification and the verification of recursive DFT for each data structure. For the recursive version of DFT, all proofs were closed. The data is also plotted as stacked bar-chart next to the table. In the chart, the red bars visualize the specification time and the blue ones the verification time. Since recursive DFT with adjacency matrix was the first proof, its specification and verification took longer than the next proofs with the adjacency array, linked or adjacency lists data structure (treated in this order). For example, about three quarters of the specification remains the same for all four data structures. This is why the specification time drops from about 10 hours to about 1 hour, after the specification was once written for the adjacency matrix version. For the following data structures, only the invariants of the used graph-representation had to be specified. Also the verification becomes easier with every new data structure, since the interactive steps are similar across all data structures. So the chart in figure 6.1 visualizes a learning curve across the four data structures.

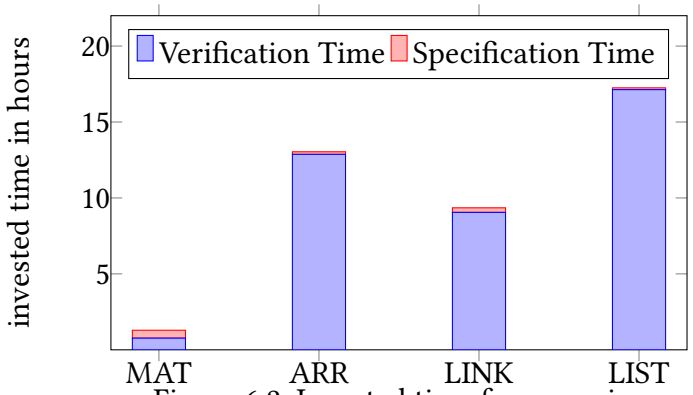


	SPEC-time	VER-time
MAT	10h 18m	76h 11m
ARR	1h 29m	64h 49m
LINK	2h 01m	40h 25m
LIST	1h 41m	40h 51m

Figure 6.1: Invested time for recursive proofs

During the verification-process of recursive DFT, after all data structures had been treated, redundant invariants, pre- and postconditions were found in the specification of DFT. After removing those redundancies, the verification was performed again for each data structure, this time with the same knowledge base for all of them. The table in figure 6.2 provides these new verification-times. Here the specification time is mostly insignificant, since the specification was already available and only some redundancies were removed. The verification time instead shows a great difference compared to the data in figure 6.1. The adjacency matrix version could be verified in less than an hour, while the adjacency array and the linked data structure required about ten hours. The adjacency lists version even took another ten hours more to verify.

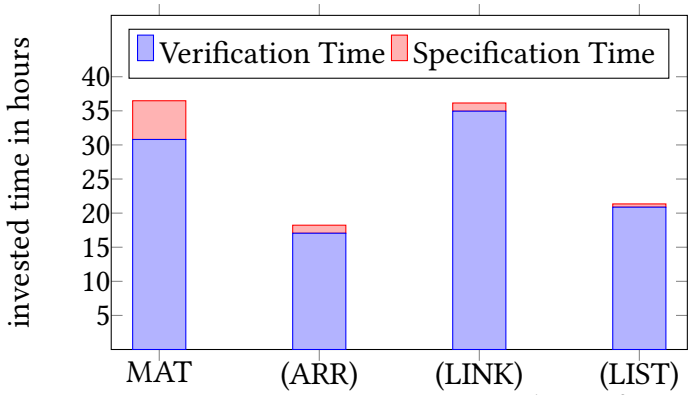
These values are not influenced by a learning curve, since they do not contain the time spent on learning to work with KeY or figuring out the required interactive steps. The plot in figure 6.2 shows how much time each data structure would need, after the interactive steps are known. In contrast to the data in figure 6.1, the data in figure 6.2 shows, how much more effort the adjacency list version requires compared to the adjacency matrix version (which is about 13 times larger).



	SPEC-time	VER-time
MAT	31m	46m
ARR	10m	12h 52m
LINK	18m	9h 03m
LIST	7m	17h 08m

Figure 6.2: Invested time for recursive proofs without learning curve

Figure 6.3 shows the time effort for the non-recursive version of DFT. For non-recursive DFT, only the adjacency matrix version is proven. The other proofs still have some open goals left. The 'Open goals'-column in figure 6.3 contains entries x/y, where x is the number of open goals that were not closed and y is the number of open goals, after applying KeY's 'Autopilot preparation' on the initial proofs, which performs symbolic execution and splits the resulting goals into multiple branches. We have to take these open goals into account, since closing them would require more time. As with the recursive version (see figure 6.1), the adjacency matrix version was the first one to be treated. Thus, there is some knowledge gained during this process. Although non-recursive DFT had already been verified with the adjacency matrix and although the proof is not closed, the linked data structure required almost the same time as the adjacency matrix proofs. The adjacency array and the linked data structure have about 2% and 7% open goals left (relative to the number of goals after preparation), while the adjacency lists version has about 15% left.



	SPEC-time	VER-time	Open goals
MAT	5h 41m	30h 48m	
(ARR)	1h 11m	17h 03m	7/327
(LINK)	1h 12m	34h 56m	25/339
(LIST)	29m	20h 52m	64/422

Figure 6.3: Invested time for non-recursive proofs

6.2 Evaluation of Proof Statistics

Additionally to comparing the working hours of each proof, they can also be compared by the number of interactive steps that were applied. Therefore we can use the proof statistics provided by KeY. A proof's statistic includes the number of rules that were applied, how

many of them were interactive steps as well as the name of each interactively applied rule. Table 6.2 shows the proof statistics of the recursive and the non-recursive proofs for all four data structures. Both tables show the number of interactive steps and the number of total rule applications for each data structure.

Table 6.2: Proof statistics for recursive (left) and non-recursive proofs (right)

REC	Total rule applications	Interactive steps
MAT	150 213	108
ARR	716 736	258
LINK	1 437 270	241
LIST	691 917	326

NONREC	Open goals	Total rule applications	Interactive steps
MAT		493 635	313
ARR	7/327	1 317 723	452
LINK	25/339	1 791 127	421
LIST	64/422	1 222 949	534

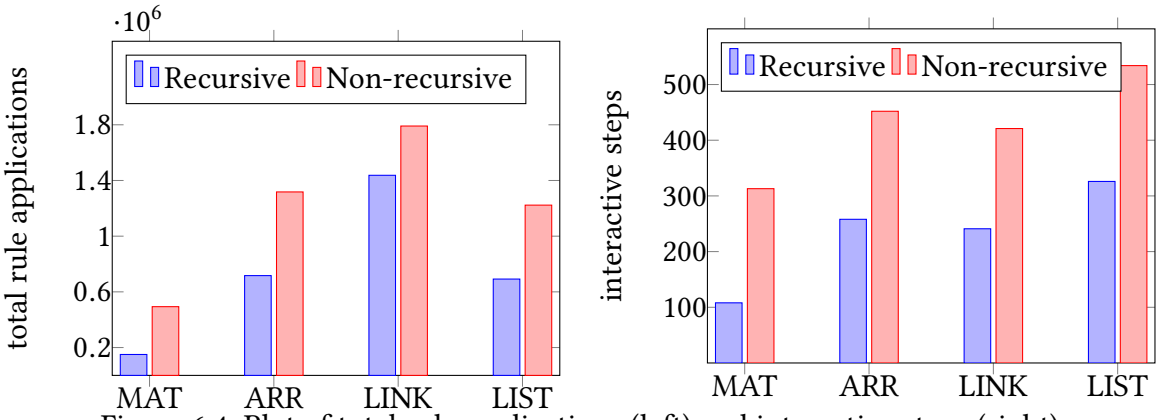


Figure 6.4: Plot of total rule applications (left) and interactive steps (right)

While all recursive proofs could be closed, there are some open goals left in the non-recursive proofs. Only the adjacency matrix version is closed, while the adjacency array, adjacency lists and the linked data structure have open goals left. Therefore, the non-recursive proof statistics contain an extra column showing the number of open goals. An entry x/y in that column describes the number of goals that were not closed (x) and the number of open goals after proof-preparation (y). By looking at the open proofs, we can find out what kind of goals are left open. For example, all three versions have open goals in the proof of `searchC`'s loop-contract for the `while`-loop (see listing 4.11). For each data structure, the corresponding proof has open goals in a branch, where the loop-contract's `assignable`-clause has to be proven. Also, the proofs of `searchC`'s inner block-contract (see listing 4.12) contain open goals regarding the `stackPtr` filed. In those goals it has to be shown that this `stackPtr` remains in certain bound, which is required to prove the algorithms termination as well as to show that the `stack`-array is not accessed out of bounds. Additionally to these open goals that appear for all three data structures, the adjacency lists and the linked data structure have open goals, where the invariants of all `List`- resp. `Vertex`-objects in `G` have to be shown to be true after each iteration of the `for`-loop. Finally, the adjacency lists version has some additional open goals. On the one hand, there are open goals, needed to prove the algorithm's termination, where an

arithmetic expression, containing a `.size()` call, has to be proven. On the other hand, when proving that the class-invariants are maintained by the loop's body, there are open goals left, in which a class-invariant, containing more than one quantifier, has to be proven. All these open goals seem to be closable, but not without additional time effort and more interactive steps.

The left chart in figure 6.4 shows the number total rule applications of the recursive proofs (blue) compared to the non-recursive number (red). Except for the linked data structure, the non-recursive proofs required almost twice as many rule applications as the recursive proofs. The right chart in figure 6.4 shows the comparison of the recursive (blue) and the non-recursive (red) form, regarding their number of interactive steps. For the adjacency array and the linked data structure, the number of interactive steps in the non-recursive version is almost twice as high as in the recursive version, although not all non-recursive proofs are closed. The non-recursive version with adjacency matrix even triples the number, needed in the recursive version. For adjacency lists, the proportion of interactive steps required for the non-recursive version to the recursive one is about two thirds. Still, we need to take into account that there around 15% goals left open, which potentially require more interactive steps.

6.3 Comparison of Recursive and Non-Recursive DFT

As the figures in sections 6.1 and 6.2 show, the non-recursive version of DFT required more effort than the recursive version. Across all four data structures, all non-recursive proofs use more rule applications and interactive steps than the recursive ones (see figure 6.4). Also, of the non-recursive proofs only the one using an adjacency matrix is closed. The other three data structures have open goals left, although the minimum time spent on the non-recursive proofs (see figure 6.3) is equal to the maximum time spent on the recursive proofs in figure 6.2. This extra effort, stems from the additional stack data structure that has to be taken care of. In the recursive implementation of DFT, the stack is implicitly present in form of the call-stack, which holds the local state for each recursive call of `searchC`. Since JML and KeY support recursion, we can use for example a `measured_by`-clause to show that the recursive version terminates, instead of arguing about the stacks size in the non-recursive version (cf. sections 3.2 and 4.2).

6.4 Comparison of Data Structures

With the evaluation of time effort and the proof statistics, we can now have a look at the advantages and disadvantages of each data structure. For each of them, we will have a general look at the (dis)advantages, followed by a paragraph in which we will look in detail, where these (dis)advantages might stem from.

First of all, the adjacency matrix was easy to work with. Its properties can be expressed in only two data structure specific invariants (see section 3.4), which are sufficient to prove both the recursive and the non-recursive version of DFT. Furthermore, the adjacency matrix did not suffer from aliasing problems during the verification process. Still, the adjacency matrix does not support the retrieval of a vertex's adjacent vertices in constant time, which is why

the implementation of DFT with this data structure requires a workaround to obtain these adjacent vertices. In terms of time effort, the adjacency matrix took the least time to verify, when leaving out the learn curve (see figure 6.2). Also in terms of total rule applications and interactive steps, it is the most efficient one.

The adjacency-matrix uses `boolean` arrays, which prevents aliasing of its entries with heap locations that are manipulated: the `int`-field `cnt` or the `int`-array `ord`'s entries. Still, when acquiring the adjacent vertices `adj = G[v]` of a vertex `v`, the `adj`-array does not only contain the adjacent vertices, but rather the information for each vertex , whether it is adjacent to `v`. This adds another check `adj[t]` to `(adj[t] && ord[t] == -1)` to find out if `t` is an unvisited adjacent vertex of `v`. This extra check resolves into an additional branch in the proof of `searchC`'s inner loop-/block-contract that has to be closed.

The adjacency array data structure was only accompanied by minor aliasing problems. Still, the adjacency arrays has more invariants than the adjacency matrix, since those properties of the adjacency array that are required to verify DFT are more complex. The time effort needed for the adjacency array data structure is similar to that of the linked data structure in the recursive version and similar to the adjacency lists time effort in the non-recursive version. Also, it requires a similar number of interactions as the linked data structure.

The adjacency array uses two `int`-arrays `G` and `adjArr`. Since an array's elements are not stored at arbitrary locations but in a fixed range on the heap, it suffices to ensure that `G`, `adjArr` and `ord` are pairwise unequal, to prevent aliasing. The invariants of this data structure are more complex than the invariants of the adjacency matrix. For example the invariant, which states that `G[i]` and `G[i+1]` are the lower and upper bound of `i`'s adjacent vertiecs in `adjArr`. This leads to interactive steps, when showing that array-accesses like `ord[adjArr[t]]` are in bounds.

The proof statistics provided by KeY (see section 6.2) contain the names of the interactively applied rules, thus the interactive steps can be divided into two classes: those that require the manual instantiation of a variable and those that do not. The linked data structure needed slightly less interactions than the adjacency array data structure. Still, the interactions for the linked data structure contain almost twice as many instantiating interactive steps as the adjacency array's interactions. Apart from that, verifying DFT with the linked data structure bought up several aliasing problems concerning the `Vertex`-objects. The non-recursive version of DFT using the linked data structure took almost the same time as the version with adjacency-matrix, regardless of the previously gained knowledge. Also that proof is still not closed, in contrast to the adjacency matrix version.

Since the linked data structure uses an extra `Vertex`-class, the invariants of all `Vertex`-objects must be shown to remain true during the verification. At those points in the proof, interactive steps were required, which is one reason why so many interactive steps were needed. When iterating, a vertex's adjacent vertices, those are `Vertex`-objects stored in a vertex's `adj`-array. As those vertices must be in the same graph, there is an invariant, which takes care that all vertices in these `adj`-arrays appear in the graph `G`. This invariant prevents those `Vertex`-objects in `adj` from aliasing and only allows them to equal one `Vertex` in `G`. The `Vertex`-class also contains an `int`-field `val` that might alias with the `cnt`-field. Still KeY could close close the

recursive proof, without an invariant preventing aliasing or interactive help.

The adjacency lists data structure required the most time to verify in its recursive version and still has around 15% open goals in its non-recursive version. Also, the proof-trees of the adjacency lists versions are less comprehensible, than those of the other versions. One reason why working with adjacency lists in its recursive version took so long is because finding missing invariants or errors in the specification was hindered by the unclear sequents. Of the four data structures, the adjacency lists caused the most aliasing problems during the verification process.

While the other data structures require only one method-call of `searchC` in `searchC`'s inner loop-/block-contract, the code of the adjacency lists version additionally contains a call of `size` and two calls of `get`. Furthermore, there are multiple `size`- and `get`-queries in the specification. Therefore, every symbolic execution of `size` or `get` splits the proof into three branches: `Pre`, `Exceptional Post` and `Post`. Thus, with every `Post`-branch, there is a new definition of the current heap. All of that makes the proof-trees and the appearing sequents less lucid. Also, a lists elements are stored at arbitrary locations on the heap, thus the invariants must assure that they do not alias with fields that are updated during DFT. The invariants concerning the other data structures are the same for the recursive and the non-recursive version. In contrast, the invariant that prevents aliasing of the lists elements must adapt to the used form of the algorithm, since in the non-recursive version of DFT, additionally the `stackPtr`-field and the entries of the `stack`-array are manipulated.

7 Conclusion

In this thesis we showed the implementations of recursive and non-recursive DFT and four graph data structures: adjacency matrix, adjacency array, linked data structure and adjacency lists. For both versions of DFT, we gave a general specification and we found invariants of the four data structures. Moreover, we looked at specific implementation, specification and verification details of all four data structures for both forms of DFT. Additionally, we showed how the depth-first property of a graph traversal can be defined in general and how it translates to JML. Finally, we evaluated the verification-processes and listed the advantages and disadvantages of the four data structures.

7.1 Results

The comparison of the graph-representations showed us that the adjacency matrix proofs took less time and required less interactive steps, than the other data structures. Also it did not suffer from aliasing problems, since it uses arrays of a different type than those fields that are manipulated by the algorithm.

The adjacency array and the linked data structure, required more time and interactive steps, as they had more complex invariants. As those invariants contained a lot of quantifiers, KeY required help at some points to instantiate them with the right variable.

Finally, we found that the adjacency lists representation required an invariant to prevent aliasing of modified fields with the lists' entries. Also, the method-calls on the lists, to obtain vertices or a list's size, resulted in a less comprehensible proof-tree.

In general, we found that the non-recursive version required more effort, than the recursive one, due to the additional stack data structure, whose behavior had to be specified and verified. The recursive version of DFT could be verified for all four graph-representations, while the non-recursive version was only closed for adjacency matrices.

Apart from comparing the data structures, we defined the depth-first property for graph traversals and used it to show that the recursive DFT implementation with adjacency matrix visits the vertices in a valid DFT-order. Also, we defined a set of taclets to treat bounded sums, deriving from `\num_of`-quantifiers.

During the verification-processes, some bugs in KeY were found and could then be fixed. For example, some proof-files reached a size of 250 megabyte. When storing such a file, a 'Java heap space OutOfMemory' error occurred, as the whole string-representation of the proof was stored in one giant array on the heap. Furthermore, the use of the `\old`-keyword in loop-contracts or nested block-contracts used to reference an undefined 'heapBeforeMethod'. These bugs were fixed and with the fixes, the verification of DFT could be completed.

7.2 Outlook

Now, that we have worked with KeY on DFT and the four graph-representations, the results could be compared to other program-verification tools and how they handle these data structures or how the data structures perform under other graph-algorithms. The gained knowledge could also be used, to verify an algorithm, which is based on DFS, like Tarjan‘s algorithm for computing strongly connected components.

Also, it could be further investigated, how this use-case can be made easier to verify in KeY.

Furthermore, the definition of the depth-first property for graph traversals could be used to verify a commonly used implementation of depth-first traversal.

Bibliography

[ABB+16] Wolfgang Ahrendt et al. *Deductive Software Verication - The KeY Book - From Theory to Practice*. Vol. LNCS 10001. Springer, 2016.

[BGU17] Bernhard Beckert, Sarah Gebring, and Mattias Ulbrich. "An Interaction Concept for Program Verification Systems with Explicit Proof Objects". In: *HVC 2017*. 2017.

[CLR+09] Thomas H. Cormen et al. "Graph Algorithms". In: *Introduction to Algorithms, Third Edition*. 2009. Chap. 6.

[Hol17] Ralf Holly. *git-autocommit*. `https://github.com/ralfholly/git-autocommit`. 2017.

[Lan18] Florian Lanzinger. "A Divide-and-Conquer Strategy with Block and Loop Contracts for Deductive Program Verification". Bachelor's Thesis. Karlsruhe Institute of Technology, 2018.

[Lar15] L. Larmore. *DFS and BFS Algorithms using Stacks and Queues*. `http://www.egr.unlv.edu/~larmore/Courses/CSC477/bfsDfs.pdf`. Accessed: 2018-08-27. 2015.

[LPC+08] Gary T. Leavens et al. *JML Reference Manual*. 2008.

[MS08] Kurt Mehlhorn and Peter Sanders. "Graph Representation". In: *Algorithms and Data Structures: The Basic Toolbox*. Jan. 2008. Chap. 8.

[MS08] Kurt Mehlhorn and Peter Sanders. "Graph Traversal". In: *Algorithms and Data Structures: The Basic Toolbox*. Jan. 2008. Chap. 9.

[MB13] L. de Moura and N. Bjørner. *Z3Prover*. `https://github.com/Z3Prover/z3`. 2013.

[RU16] Phillip Rümmer and Mattias Ulbrich. "Proof Search with Taclets". In: Wolfgang Ahrendt et al. *Deductive Software Verication - The KeY Book - From Theory to Practice*. Vol. LNCS 10001. Springer, 2016. Chap. 4.1.

[SeSc03] Robert Sedgewick and Michael Schidlowsky. *Algorithms in Java, Part 5: Graph Algorithms*. 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0201361213.

[Wac12] Simon Wacker. "Blockverträge". Studienarbeit. Karlsruher Institut für Technologie, 2012.

A Appendix

Algorithm 1: Recursive DFS

```
Data: G = (V, E) directed graph
begin dfs_init
    Visited ⟵ ∅
    foreach v ∈ V do
        dfs(v)

Data: G = (V, E) directed graph and vertex v ∈ V
begin dfs
    if v ∉ Visited then
        Visited ⟵ Visited ∪ {v}
        foreach (v, u) ∈ E do
            dfs(u)
```

Algorithm 2: Non-Recursive DFS

```
Data: G = (V, E) directed graph
begin dfs_init
    Visited ⟵ ∅
    foreach v ∈ V do
        stack_dfs(v)

Data: G = (V, E) directed graph and vertex v ∈ V
begin stack_dfs
    S ⟵ ∅
    S.push(v)

    while S ≠ ∅ do
        u ⟵ S.pop()
        if u ∉ Visited then
            Visited ⟵ Visited ∪ {u}
            foreach (u, w) ∈ E do
                if w ∉ Visited then
                    S.push(w)
```

```
\lemma
bsum_num_of_bounds2 {
  \find(bsum{uSub;} (i0, i2, \if(phi)\then(0)\else(1)))
  \varcond ( \notFreeIn(uSub, i0),
  \notFreeIn(uSub, i2))
  \add(    0 <= bsum{uSub;} (i0, i2, \if(phi)\then(0)\else(1)),
     i0 <= i2 -> bsum{uSub;} (i0, i2, \if(phi)\then(0)\else(1)) <= i2 - i0 ==>)
};
```

Listing A.1: Bsum-taclet: min- and maimum value of num_of-bsum

```
\lemma
bsum_num_of_is_max3 {
  \find(bsum{uSub;} (i0, i2, \if(phi)\then(0)\else(1)) = i2 - i0 ==>)
  \varcond ( \notFreeIn(uSub, i0),
  \notFreeIn(uSub, i2))
  \add(\forall uSub; ((uSub>=i0 & uSub<i2) -> !phi)==>)
};
```

Listing A.2: Bsum-taclet: num_of-bsum equals maximum value

```
\lemma
bsum_num_of_lt_max3 {
  \assumes(i2 > i0 & i0 >= 0 ==>)
  \find(bsum{uSub;} (i0, i2, \if(phi)\then(0)\else(1)) < i2 - i0 ==>)
  \varcond ( \notFreeIn(uSub, i0),
  \notFreeIn(uSub, i2))
  \add(\exists uSub; ((uSub>=i0 & uSub<i2) & phi)==>)
};
```

Listing A.3: Bsum-taclet: num_of-bsum less then maximum value

```
public class GraphDFS_dft_prop {

  /*@ public invariant (\forall int j; 0 <= j && j < G.length;
                        G[j] != null && G[j].length == G.length);
    @ public invariant G.length > 0;
    @*/
  private /*@ spec_public @*/ boolean[][] G;

  /*@ public invariant 0 <= cnt && cnt <= G.length;
    @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
    @*/
  private /*@ spec_public @*/ int cnt;

  /*@ public invariant ord.length == G.length;
    @ public invariant (\forall int i; 0 <= i && i < cnt;
                         (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @ public invariant (\forall int i; 0 <= i && i < ord.length;
                         -1 <= ord[i] && ord[i] < cnt);
    @*/
  private /*@ spec_public @*/ int[] ord;

  /*@ public invariant t.length == G.length && ord != t;
    @ public invariant (\forall int i; 0 <= i && i < cnt;
                          0 <= t[i] && t[i] < G.length && ord[t[i]] == i);
    @ public invariant (\forall int i; 0 <= i && i < G.length;
                          (ord[i] != -1) ==> t[ord[i]] == i);
    @ public invariant (\forall int i; cnt <= i && i < G.length; t[i] == -1);
    @ public invariant (\forall int v; 0 <= v && v < cnt;
                         (\exists int p; -1 <= p && p < v;
                             (G[t[p]][t[v]] || p == -1)
                            && (\forall int w; p < w && w < v;
                                 (\forall int y; 0 <= y && y < G.length;
                                     G[t[w]][y] ==> (0 <= ord[y] && ord[y] < v))) ));
    @*/
  /*@ ghost int[] t;
    @*/
```

```
/*@ public normal_behavior
  @  requires cnt < G.length;
  @  requires 0 <= v && v < G.length;
  @  requires ord[v] == -1;
  @  requires (\exists int p; -1 <= p && p < cnt;
                 (G[t[p]][v] || p == -1)
               && (\forall int w; p < w && w < cnt;
                     (\forall int y; 0 <= y && y < G.length;
                        G[t[w]][y] ==> (0 <= ord[y] && ord[y] < cnt))) );
  @  ensures cnt > \old(cnt);
  @  ensures (ord[v] == \old(cnt)) && (ord[v] != -1);
  @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @  ensures (\forall int i; ord[v] <= i && i < cnt;
                (\forall int y; 0 <= y && y < G.length;
                       G[t[i]][y] ==> (0 <= ord[y] && ord[y] < cnt)));
  @  measured_by (G.length - cnt);
  @ assignable unvisited(), cnt, t[cnt .. t.length];
  @*/
public void searchC(int v) {
  /*@ set t[cnt] = v;
    @*/
  ord[v] = cnt++;
  boolean[] adj = G[v];

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires adj == G[v] && 0 <= v && v < G.length;
    @  requires 0 <= t && t <= adj.length;
    @  requires ord[v] == \old(cnt);
    @  requires 0 <= \old(cnt) && \old(cnt) < cnt && \old(G.length) == G.length;
    @  requires (\forall int i; ord[v] < i && i < cnt;
                   (\forall int y; 0 <= y && y < G.length;
                      G[t[i]][y] ==> (0 <= ord[y] && ord[y] < cnt)));
    @  requires (\forall int y; 0 <= y && y < t;
                   G[v][y] ==> (0 <= ord[y] && ord[y] < cnt));
    @  ensures ord[v] != -1 && \before(cnt) <= cnt;
    @  ensures \invariant_for(this);
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                  (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @  ensures (\forall int i; ord[v] <= i && i < cnt; #
                  (\forall int y; 0 <= y && y < G.length;
                        G[t[i]][y] ==> (0 <= ord[y] && ord[y] < cnt)));
    @ decreases adj.length - t;
    @ measured_by (G.length - cnt);
    @ assignable unvisited(), cnt, t, t[cnt .. t.length];
    @*/
```

```
  {
    for (int t = 0; t < adj.length; t++) {
      if (adj[t] && ord[t] == -1) {searchC(t);}
    }
  }

}

/*@ public normal_behavior
  @ requires (\forall int j; 0 <= j && j < GIn.length; GIn[j].length == GIn.length);
  @ requires GIn.length > 0;
  @
  @ ensures G == GIn;
  @ ensures (cnt == 0);
  @ ensures (\forall int i; 0 <= i && i < G.length; ord[i] == -1);
  @ ensures (\forall int i; 0 <= i && i < G.length; t[i] == -1);
  @*/
public GraphDFS_dft_prop(boolean[][] GIn) {
  this.G = GIn;
  cnt = 0;
  /*@ set t = new int[G.length];
  @*/
  ord = new int[G.length];

  /*@ loop_invariant 0 <= r && r <= G.length;
    @ loop_invariant (\forall int i; 0 <= i && i < r; ord[i] == -1);
    @ loop_invariant (\forall int i; 0 <= i && i < r; t[i] == -1);
    @ decreases ord.length - r;
    @ assignable ord[r .. (ord.length - 1)], t[r .. (t.length - 1)], r;
    @*/
  for (int r = 0; r < G.length; r++) {
    /*@ set t[r] = -1;
    @*/
    ord[r] = -1;
  }

}

/*@ public normal_behavior
  @  requires cnt == 0;
  @  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
  @  ensures (cnt == G.length);
  @  ensures (\forall int i; 0 <= i && i < G.length; ord[i] != -1);
  @  ensures (\forall int i; 0 <= i && i < G.length;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @ assignable ord[*], cnt, t[*];
  @*/
```

```
public void dfs() {

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires 0 <= k && k <= G.length;
    @  requires (\forall int i; 0 <= i && i < k; ord[i] != -1);
    @  requires cnt >= k;
    @  requires (\forall int i; 0 <= i && i < cnt;
                   (\forall int y; 0 <= y && y < G.length;
                          G[t[i]][y] ==> (0 <= ord[y] && ord[y] < cnt)));
    @
    @  ensures (ord[k-1] != -1);
    @  ensures (\forall int i; 0 <= i && i < k; ord[i] != -1);
    @  ensures ord[k-1] != -1;
    @  ensures cnt >= k;
    @  ensures k == G.length;
    @  ensures \invariant_for(this);
    @
    @ decreases G.length - k;
    @ assignable unvisited(), cnt, k, t[*];
    @*/
  {
    for (int k = 0; k < G.length; k++) {
      if (ord[k] == -1) {
        searchC(k);
      }
    }

  }
}

/*@ model \locset unvisited() {
       return \infinite_union(int i; (0 <= i && i < ord.length && ord[i] == -1);
           \singleton(ord[i]));
} @*/

/*@ public normal_behavior
  @  requires 0 <= v && v < G.length;
  @  ensures \result == ord[v];
  @  assignable \strictly_nothing;
  @*/
public int order(int v) {
  return ord[v];
}
```

```
  /*@ public normal_behavior
    @  ensures \result == cnt;
    @  assignable \strictly_nothing;
    @*/
  public int count() {
    return cnt;
  }

}
```

Listing A.4: Recursive DFT using an adjacency matrix with depth-first property invariant

```
public class GraphDFS_matr {

  /*@ public invariant (\forall int j; 0 <= j && j < G.length; G[j] != null &&
      G[j].length == G.length);
    @ public invariant G.length > 0;
    @*/
  private /*@ spec_public @*/ boolean[][] G;

  /*@ public invariant 0 <= cnt && cnt <= G.length;
    @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
    @*/
  private /*@ spec_public @*/ int cnt;

  /*@ public invariant ord.length == G.length;
    @ public invariant (\forall int i; 0 <= i && i < cnt;
                         (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @*/
  private /*@ spec_public @*/ int[] ord;




  /*@ public normal_behavior
    @  requires cnt < G.length;
    @  requires 0 <= v && v < G.length;
    @  requires ord[v] == -1;
    @
    @  ensures cnt > \old(cnt);
    @  ensures (ord[v] == \old(cnt)) && (ord[v] != -1);
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                   (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @  measured_by (G.length - cnt);
    @ assignable unvisited(), cnt;
    @*/
```

```
public void searchC(int v) {
  ord[v] = cnt++;
  boolean[] adj = G[v];

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires adj == G[v] && 0 <= v && v < G.length;
    @  requires 0 <= t && t <= adj.length;
    @  requires ord[v] != -1 ;
    @  requires 0 <= \old(cnt) && \old(cnt) < cnt && \old(G.length) == G.length;
    @  ensures ord[v] != -1 && \before(cnt) <= cnt;
    @  ensures \invariant_for(this);
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                  (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @ decreases adj.length - t;
    @ measured_by (G.length - cnt);
    @ assignable unvisited(), cnt, t;
    @*/
  {
    for (int t = 0; t < adj.length; t++) {
      if (adj[t] && ord[t] == -1) {searchC(t);}
    }
  }

}

/*@ public normal_behavior
  @ requires (\forall int j; 0 <= j && j < GIn.length; GIn[j].length == GIn.length);
  @ requires GIn.length > 0;
  @ ensures G == GIn;
  @ ensures (cnt == 0);
  @ ensures (\forall int i; 0 <= i && i < G.length; ord[i] == -1);
  @*/
public GraphDFS_matr(boolean[][] GIn) {
  this.G = GIn;
  cnt = 0;
  ord = new int[G.length];

  /*@ loop_invariant 0 <= t && t <= G.length;
    @ loop_invariant (\forall int i; 0 <= i && i < t; ord[i] == -1);
    @ decreases ord.length - t;
    @ assignable ord[t .. (ord.length - 1)], t;
    @*/
  for (int t = 0; t < G.length; t++) {
    ord[t] = -1;
  }
}
```

```
/*@ public normal_behavior
@  requires cnt == 0;
@  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
@
@  ensures (cnt == G.length);
@  ensures (\forall int i; 0 <= i && i < G.length; ord[i] != -1);
@  ensures (\forall int i; 0 <= i && i < G.length;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
@ assignable ord[*], cnt;
@*/
public void dft() {

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires 0 <= k && k <= G.length;
    @  requires (\forall int i; 0 <= i && i < k; ord[i] != -1);
    @  requires cnt >= k;
    @
    @  ensures (ord[k-1] != -1);
    @  ensures (\forall int i; 0 <= i && i < k; ord[i] != -1);
    @  ensures ord[k-1] != -1;
    @  ensures cnt >= k;
    @  ensures k == G.length;
    @  ensures \invariant_for(this);
    @
    @ decreases G.length - k;
    @ assignable unvisited(), cnt, k;
    @*/
  {
    for (int k = 0; k < G.length; k++) {
      if (ord[k] == -1) {searchC(k);}
    }
  }
}

/*@ model \locset unvisited() {
        return \infinite_union(int i; (0 <= i && i < ord.length && ord[i] == -1);
            \singleton(ord[i]));
} @*/


/*@ public normal_behavior
  @  requires 0 <= v && v < G.length;
  @  ensures \result == ord[v];
  @  assignable \strictly_nothing;
  @*/
```

```
  public int order(int v) {
    return ord[v];
  }

  /*@ public normal_behavior
    @  ensures \result == cnt;
    @  assignable \strictly_nothing;
    @*/
  public int count() {
    return cnt;
  }

}
```

Listing A.5: Recursive DFT using an adjacency matrix

```
  public class GraphDFS_adjarr {

  /*@ public invariant (\forall int j; 0 <= j && j < len;
                        0 <= G[j] && G[j] < adjArr.length && (G[j] <= G[j+1]));
    @ public invariant len > 0;
    @ public invariant G != adjArr && adjArr != ord && ord != G;
    @ public invariant 0 <= G[len] && G[len] < adjArr.length;
    @*/
  private /*@ spec_public @*/ int[] G;

  /*@ model int len;
    @ represents len \such_that len == G.length - 1;
    @*/

  /*@ public invariant (\forall int i; 0 <= i && i < adjArr.length;
                        0 <= adjArr[i] && adjArr[i] < len);
    @*/
  private /*@ spec_public @*/ int[] adjArr;

  /*@ public invariant 0 <= cnt && cnt <= len;
    @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
    @*/
  private /*@ spec_public @*/ int cnt;

  /*@ public invariant ord.length == len;
    @ public invariant (\forall int i; 0 <= i && i < cnt;
                         (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @*/
  private /*@ spec_public @*/ int[] ord;
```

```
/*@ public normal_behavior
  @  requires cnt < len;
  @  requires 0 <= v && v < len;
  @  requires ord[v] == -1;
  @
  @  ensures cnt > \old(cnt);
  @  ensures (ord[v] == \old(cnt)) && (ord[v] != -1);
  @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @  measured_by (len - cnt);
  @ assignable unvisited(), cnt;
  @*/
public void searchC(int v) {

  ord[v] = cnt++;

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires G[v] <= t && t <= G[v+1];
    @  requires ord[v] != -1 ;
    @  requires 0 <= \old(cnt) && \old(cnt) < cnt && \old(len) == len;
    @  requires 0 <= v && v < len;
    @
    @  ensures ord[v] != -1 && \before(cnt) <= cnt;
    @  ensures \invariant_for(this);
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                  (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @ decreases G[v+1] - t;
    @ measured_by (len - cnt);
    @ assignable unvisited(), cnt, t;
    @*/
  {
    for (int t = G[v]; t < G[v+1]; t++) {
      if (ord[adjArr[t]] == -1) {searchC(adjArr[t]);}
    }
  }

}
```

```
/*@ public normal_behavior
  @ requires GIn.length - 1 > 0;
  @ requires (\forall int j; 0 <= j && j < (GIn.length - 1);
                0 <= GIn[j] && GIn[j] < adjIn.length && (GIn[j] <= GIn[j+1]));
  @ requires (\forall int i; 0 <= i && i < adjIn.length;
                0 <= adjIn[i] && adjIn[i] < (GIn.length - 1));
  @ requires GIn != adjIn;
  @ requires 0 <= GIn[GIn.length - 1] && GIn[GIn.length - 1] < adjIn.length;
  @
  @ ensures (cnt == 0);
  @ ensures (\forall int i; 0 <= i && i < G.length - 1; ord[i] == -1);
  @*/
public GraphDFS_adjarr(int[] GIn, int[] adjIn) {
  this.G = GIn;
  this.adjArr = adjIn;
  cnt = 0;
  ord = new int[G.length - 1];

  /*@ loop_invariant 0 <= t && t <= len;
    @ loop_invariant (\forall int i; 0 <= i && i < t; ord[i] == -1);
    @ decreases ord.length - t;
    @ assignable ord[t .. (ord.length - 1)], t;
    @*/
  for (int t = 0; t < G.length - 1; t++) {
    ord[t] = -1;
  }

}
```

```
/*@ public normal_behavior
  @  requires cnt == 0;
  @  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
  @  ensures (cnt == len);
  @  ensures (\forall int i; 0 <= i && i < ord.length; ord[i] != -1);
  @  ensures (\forall int i; 0 <= i && i < ord.length;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @ assignable ord[*], cnt;
  @*/
public void dft() {
```

```
    /*@ loop_contract normal_behavior
      @  requires \invariant_for(this);
      @  requires 0 <= k && k <= len;
      @  requires (\forall int i; 0 <= i && i < k; ord[i] != -1);
      @  requires cnt >= k;
      @  ensures (ord[k-1] != -1);
      @  ensures (\forall int i; 0 <= i && i < k; ord[i] != -1);
      @  ensures ord[k-1] != -1;
      @  ensures cnt >= k;
      @  ensures k == len;
      @  ensures \invariant_for(this);
      @
      @ decreases len - k;
      @ assignable unvisited(), cnt, k;
      @*/
    {
      for (int k = 0; k < G.length - 1; k++) {
        if (ord[k] == -1) {searchC(k);}
      }
    }
  }

  /*@ model \locset unvisited() {
          return \infinite_union(int i; (0 <= i && i < ord.length && ord[i] == -1);
              \singleton(ord[i]));
  } @*/

  /*@ public normal_behavior
    @  requires 0 <= v && v < len;
    @  ensures \result == ord[v];
    @  assignable \strictly_nothing;
    @*/
  public int order(int v) {
    return ord[v];
  }

  /*@ public normal_behavior
    @  ensures \result == cnt;
    @  assignable \strictly_nothing;
    @*/
  public int count() {
    return cnt;
  }

}
```

Listing A.6: Recursive DFT using an adjacency array

```
public class GraphDFS_linked {

  /*@ public invariant (\forall int j; 0 <= j && j < G.length;
                        G[j] != null && G[j].adj.length <= G.length);
    @ public invariant G.length > 0;
    @ public invariant (\forall int i; 0 <= i && i < G.length;
                         (\forall int j; 0 <= j && j < G[i].adj.length;
                                0 <= G[i].adj[j].val && G[i].adj[j].val < G.length
                               && G[i].adj[j] == G[G[i].adj[j].val]));
    @ public invariant (\forall int i; 0 <= i && i < G.length; G[i].val == i);
    @ public invariant (\forall int i; 0 <= i && i < G.length;
                         (\forall int j; 0 <= j && j < G.length; (i!=j) ==> (G[i] !=
                             G[j]) ));
    @ public invariant (\forall int i; 0 <= i && i < G.length; \invariant_for(G[i]));
    @*/
  private /*@ spec_public @*/ Vertex[] G;

  /*@ public invariant 0 <= cnt && cnt <= G.length;
    @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
    @*/
  private /*@ spec_public @*/ int cnt;

  /*@ public invariant ord.length == G.length;
    @ public invariant (\forall int i; 0 <= i && i < cnt;
                         (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @*/
  private /*@ spec_public @*/ int[] ord;









  /*@ public normal_behavior
    @  requires cnt < G.length;
    @  requires 0 <= v && v < G.length;
    @  requires ord[v] == -1;
    @
    @  ensures cnt > \old(cnt);
    @  ensures (ord[v] == \old(cnt)) && (ord[v] != -1);
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                    (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @  measured_by (G.length - cnt);
    @ assignable unvisited(), cnt;
    @*/
```

```
public void searchC(int v) {

  ord[v] = cnt++;
  Vertex[] adj = G[v].adj;

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires adj == G[v].adj && 0 <= v && v < G.length;
    @  requires 0 <= t && t <= adj.length;
    @  requires ord[v] != -1;
    @  requires 0 <= \old(cnt) && \old(cnt) < cnt && \old(G.length) == G.length;
    @
    @  ensures ord[v] != -1 && \before(cnt) <= cnt;
    @  ensures \invariant_for(this);
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                   (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @ decreases adj.length - t;
    @ measured_by (G.length - cnt);
    @ assignable unvisited(), cnt, t;
    @*/
  {
    for (int t = 0; t < adj.length; t++) {
      if (ord[adj[t].val] == -1) {searchC(adj[t].val);}
    }
  }

}


/*@ public normal_behavior
  @ requires (\forall int j; 0 <= j && j < GIn.length;
                GIn[j].adj.length <= GIn.length);
  @ requires GIn.length > 0;
  @ requires (\forall int i; 0 <= i && i < GIn.length;
              (\forall int j; 0 <= j && j < GIn[i].adj.length;
                    0 <= GIn[i].adj[j].val && GIn[i].adj[j].val < GIn.length
                 && GIn[i].adj[j] == GIn[GIn[i].adj[j].val]));
  @ requires (\forall int i; 0 <= i && i < GIn.length; GIn[i].val == i);
  @ requires (\forall int i; 0 <= i && i < GIn.length;
                (\forall int j; 0 <= j && j < GIn.length;
                      (i!=j) ==> (GIn[i] != GIn[j]) ));
  @ requires (\forall int i; 0 <= i && i < GIn.length; \invariant_for(GIn[i]));
  @
  @ ensures G == GIn;
  @ ensures (cnt == 0);
  @ ensures (\forall int i; 0 <= i && i < G.length; ord[i] == -1);
  @*/
```

```
  public GraphDFS_linked(Vertex[] GIn) {
    this.G = GIn;
    cnt = 0;
    ord = new int[G.length];

    /*@ loop_invariant 0 <= t && t <= G.length;
      @ loop_invariant (\forall int i; 0 <= i && i < t; ord[i] == -1);
      @ decreases ord.length - t;
      @ assignable ord[t .. (ord.length - 1)], t;
      @*/
    for (int t = 0; t < G.length; t++) {
      ord[t] = -1;
    }

  }

  /*@ public normal_behavior
    @  requires cnt == 0;
    @  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
    @  ensures (cnt == G.length);
    @  ensures (\forall int i; 0 <= i && i < G.length; ord[i] != -1);
    @  ensures (\forall int i; 0 <= i && i < G.length;
                  (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @ assignable ord[*], cnt;
    @*/
  public void dft() {

    /*@ loop_contract normal_behavior
      @  requires \invariant_for(this);
      @  requires 0 <= k && k <= G.length;
      @  requires (\forall int i; 0 <= i && i < k; ord[i] != -1);
      @  requires cnt >= k;
      @  ensures (\forall int i; 0 <= i && i < k; ord[i] != -1);
      @  ensures ord[k-1] != -1;
      @  ensures cnt >= k;
      @  ensures k == G.length;
      @  ensures \invariant_for(this);
      @
      @ decreases G.length - k;
      @ assignable unvisited(), cnt, k;
      @*/
    {
      for (int k = 0; k < G.length; k++) {
        if (ord[k] == -1) {searchC(k);}
      }
    }
  }
```

```
  /*@ model \locset unvisited() {
          return \infinite_union(int i; (0 <= i && i < ord.length && ord[i] == -1);
              \singleton(ord[i]));
  } @*/

  /*@ public normal_behavior
    @  requires 0 <= v && v < G.length;
    @  ensures \result == ord[v];
    @  assignable \strictly_nothing;
    @*/
  public int order(int v) {
    return ord[v];
  }

  /*@ public normal_behavior
    @  ensures \result == cnt;
    @  assignable \strictly_nothing;
    @*/
  public int count() {
    return cnt;
  }

}
```

Listing A.7: Recursive DFT using a linked data structure

```
public final class Vertex {

  /*@ public invariant val >= 0; @*/
  int val;

  /*@ public invariant adj != null;
    @ public invariant (\forall int i; 0 <= i && i < adj.length; adj[i] != null);
    @ public invariant (\forall int i; 0 <= i && i < adj.length;
                        (\forall int j; 0 <= j && j < adj.length;
                                (i==j) || (adj[i] != adj[j] && adj[i].val !=
                                    adj[j].val)));
    @*/
  Vertex[] adj;
}
```

Listing A.8: Vertex class

```
public class GraphDFS_adjlists {

  /*@ public invariant (\forall int j; 0 <= j && j < G.length;
                        G[j] != null && G[j].size() <= G.length);
    @ public invariant G.length > 0;
    @ public invariant (\forall int i; 0 <= i && i < G.length;
                         (\forall int j; 0 <= j && j < G[i].size();
                             0 <= G[i].get(j) && G[i].get(j) < G.length));
    @ public invariant (\forall int i; 0 <= i && i < G.length; \invariant_for(G[i]));
    @ public invariant (\forall int i; 0 <= i && i < G.length;
        \disjoint(G[i].footprint,\singleton(cnt)) && (\forall int j; 0 <= j && j <
        ord.length; \disjoint(G[i].footprint,\singleton(ord[j])) ));
    @*/
  private /*@ spec_public @*/ List[] G;

  /*@ public invariant 0 <= cnt && cnt <= G.length;
    @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
    @*/
  private /*@ spec_public @*/ int cnt;

  /*@ public invariant ord.length == G.length;
    @ public invariant (\forall int i; 0 <= i && i < cnt;
                         (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @*/
  private /*@ spec_public @*/ int[] ord;

  /*@ public normal_behavior
  @  requires cnt < G.length;
  @  requires 0 <= v && v < G.length;
  @  requires ord[v] == -1;
  @
  @  ensures cnt > \old(cnt);
  @  ensures (ord[v] == \old(cnt)) && (ord[v] != -1);
  @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                 (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @  measured_by (G.length - cnt);
  @ assignable unvisited(), cnt;
  @*/
```

```
public void searchC(int v) {

  ord[v] = cnt++;
  List adj = G[v];

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires adj == G[v] && 0 <= v && v < G.length;
    @  requires 0 <= t && t <= adj.size();
    @  requires ord[v] != -1 ;
    @  requires 0 <= \old(cnt) && \old(cnt) < cnt && \old(G.length) == G.length;
    @
    @  ensures ord[v] != -1 && \before(cnt) <= cnt;
    @  ensures \invariant_for(this);
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                   (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @ decreases adj.size() - t;
    @ measured_by (G.length - cnt);
    @ assignable unvisited(), cnt, t;
    @*/
  {
    for (int t = 0; t < adj.size(); t++) {
      if (ord[adj.get(t)] == -1) {searchC(adj.get(t));}
    }
  }

}
```

```
/*@ public normal_behavior
  @ requires (\forall int j; 0 <= j && j < GIn.length;
                GIn[j].size() <= GIn.length && \invariant_for(GIn[j]));
  @ requires GIn.length > 0;
  @ requires (\forall int i; 0 <= i && i < GIn.length;
                (\forall int j; 0 <= j && j < GIn[i].size();
                    0 <= GIn[i].get(j) && GIn[i].get(j) < GIn.length));
  @ requires (\forall int i; 0 <= i && i < GIn.length; \invariant_for(GIn[i]));
  @
  @ ensures G == GIn;
  @ ensures (cnt == 0);
  @ ensures (\forall int i; 0 <= i && i < G.length; ord[i] == -1);
  @*/
```

```
  public GraphDFS_adjlists(List[] GIn) {
    this.G = GIn;
    cnt = 0;
    ord = new int[G.length];

    /*@ loop_invariant 0 <= t && t <= G.length;
      @ loop_invariant (\forall int i; 0 <= i && i < t; ord[i] == -1);
      @ decreases ord.length - t;
      @ assignable ord[t .. (ord.length - 1)], t;
      @*/
    for (int t = 0; t < G.length; t++) {
      ord[t] = -1;
    }

  }

  /*@ public normal_behavior
    @  requires cnt == 0;
    @  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
    @  ensures (cnt == G.length);
    @  ensures (\forall int i; 0 <= i && i < G.length; ord[i] != -1);
    @  ensures (\forall int i; 0 <= i && i < G.length;
                  (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @ assignable ord[*], cnt;
    @*/
  public void dfs() {

    /*@ loop_contract normal_behavior
      @  requires \invariant_for(this);
      @  requires 0 <= k && k <= G.length;
      @  requires (\forall int i; 0 <= i && i < k; ord[i] != -1);
      @  requires cnt >= k;
      @  ensures (ord[k-1] != -1);
      @  ensures (\forall int i; 0 <= i && i < k; ord[i] != -1);
      @  ensures ord[k-1] != -1;
      @  ensures cnt >= k;
      @  ensures k == G.length;
      @  ensures \invariant_for(this);
      @ decreases G.length - k;
      @ assignable unvisited(), cnt, k;
      @*/
    {
      for (int k = 0; k < G.length; k++) {
        if (ord[k] == -1) {searchC(k);}
      }
    }
  }
```

```
  /*@ model \locset unvisited() {
          return \infinite_union(int i; (0 <= i && i < ord.length && ord[i] == -1);
              \singleton(ord[i]));
  } @*/

  /*@ public normal_behavior
    @  requires 0 <= v && v < G.length;
    @  ensures \result == ord[v];
    @  assignable \strictly_nothing;
    @*/
  public int order(int v) {
    return ord[v];
  }

  /*@ public normal_behavior
    @  ensures \result == cnt;
    @  assignable \strictly_nothing;
    @*/
  public int count() {
    return cnt;
  }

}
```

Listing A.9: Recursive DFT using a adjacency lists

```
public interface List {

  //@ public ghost instance \locset footprint;
  //@ public ghost instance \seq seq;

  //@ public instance invariant \subset(\singleton(this.seq), footprint);
  //@ public instance invariant \subset(\singleton(this.footprint), footprint);
  //@ public instance invariant (\forall int i; 0<=i && i<seq.length;
  //                              ((int)seq[i]) != null); //type(seq[i]) = type(int)
  //@ public accessible \inv: footprint;

  /*@ public normal_behaviour
    @  accessible footprint;
    @  ensures \result == seq.length;
    @*/
  public /*@pure@*/ int size();
```

```
  /*@ public normal_behaviour
    @  requires 0 <= index && index < seq.length;
    @  accessible footprint;
    @  ensures \result == seq[index];
    @
    @ also public exceptional_behaviour
    @  requires index < 0 || seq.length <= index;
    @  signals_only IndexOutOfBoundsException;
    @*/
  public /*@pure@*/ int get(int index);

  /*...*/
}
```

Listing A.10: A simplified List interface

```
public class GraphDFS_matr_nonrec {
  /*@ public invariant (\forall int j; 0 <= j && j < G.length;
                        G[j] != null && G[j].length == G.length);
    @ public invariant G.length > 0;
    @*/
  boolean[][] G;

  /*@ public invariant 0 <= cnt && cnt <= G.length;
    @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
    @*/
  int cnt;

  /*@ public invariant ord.length == G.length;
    @ public invariant (\forall int i; 0 <= i && i < cnt;
                        (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @*/
  int[] ord;

  /*@ public invariant 0 <= stackPtr && stackPtr <= stack.length
                                     && stackPtr <= (cnt + 1)*G.length;
    @*/
  int stackPtr;

  /*@ public invariant stack.length == (G.length * G.length);
    @ public invariant (\forall int i; 0 <= i && i < stackPtr;
                        0 <= stack[i] && stack[i] < G.length);
    @ public invariant (\forall int i; stackPtr <= i && i < stack.length;
                        stack[i] == -1);
    @ public invariant stack != ord;
    @*/
  int[] stack;
```

```
/*@ public normal_behavior
  @ requires (\forall int j; 0 <= j && j < GIn.length; GIn[j].length == GIn.length);
  @ requires GIn.length > 0;
  @
  @ ensures G == GIn;
  @ ensures stackPtr == 0;
  @ ensures (cnt == 0);
  @ ensures (\forall int j; 0 <= j && j < ord.length; ord[j] == -1);
  @ ensures (\forall int j; 0 <= j && j < stack.length; stack[j] == -1);
  @*/
public GraphDFS_matr_nonrec(boolean[][] GIn) {
  G = GIn;
  cnt = 0;
  ord = new int[G.length];

  stackPtr = 0;
  stack = new int[G.length * G.length];

  /*@ loop_invariant 0 <= i && i <= ord.length;
    @ loop_invariant (\forall int j; 0 <= j && j < i; ord[j] == -1);
    @ decreases ord.length - i;
    @ assignable ord[i .. (ord.length - 1)], i;
    @*/
  for(int i = 0; i < ord.length; i++) {
    ord[i] = -1;
  }

  /*@ loop_invariant 0 <= k && k <= stack.length;
    @ loop_invariant (\forall int j; 0 <= j && j < k; stack[j] == -1);
    @ decreases stack.length - k;
    @ assignable stack[k .. (stack.length - 1)], k;
    @*/
  for(int k = 0; k < stack.length; k++) {
    stack[k] = -1;
  }
}

/*@ model \locset unvisited() {
        return \infinite_union(int i; (0 <= i && i < ord.length && ord[i] == -1);
            \singleton(ord[i]));
} @*/
```

```
/*@ public normal_behavior
  @  requires cnt == 0 && stackPtr == 0;
  @  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
  @
  @  ensures (cnt == G.length);
  @  ensures (\forall int i; 0 <= i && i < G.length; ord[i] != -1);
  @  ensures (\forall int i; 0 <= i && i < G.length;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @ assignable ord[*], cnt, before, diff, stack[*], stackPtr;
  @*/
public void dft() {

  /*@ loop_invariant 0 <= k && k <= G.length;
    @ loop_invariant \invariant_for(this);
    @ loop_invariant (\forall int i; 0 <= i && i < k; ord[i] != -1);
    @ loop_invariant cnt >= k && stackPtr == 0;
    @ decreases G.length - k;
    @ assignable unvisited(), cnt, k, before, diff, stack[*], stackPtr;
    @*/
  for(int k = 0; k < G.length; k++) {
    if (ord[k] == -1){searchC(k);}
  }
}

/*@ ghost int diff;
  @ ghost int before;
  @*/
```

```
/*@ public normal_behavior
  @  requires cnt < G.length;
  @  requires 0 <= v && v < G.length;
  @  requires ord[v] == -1;
  @  requires stackPtr == 0;
  @
  @  ensures cnt > \old(cnt);
  @  ensures (ord[v] == \old(cnt)) && (ord[v] != -1);
  @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @  ensures stackPtr == 0;
  @ assignable unvisited(), cnt, stack[*], stackPtr, before, diff;
  @*/
```

```
public void searchC(int v) {
  stack[stackPtr++] = v;

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires 0 <= v && v < G.length;
    @  requires (stack[0] == v && ord[v] == -1 && stackPtr == 1 && cnt == \old(cnt))
          || (cnt > \old(cnt) && ord[v] == \old(cnt));
    @  requires cnt <= G.length;
    @  requires 0 <= \old(cnt) && \old(cnt) <= G.length;
    @
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @  ensures \old(cnt) < cnt;
    @  ensures ord[v] == \old(cnt);
    @  ensures stackPtr == 0;
    @  ensures \invariant_for(this);
    @ decreases (G.length * G.length) - (cnt * G.length) + stackPtr;
    @ assignable stackPtr, stack[*], cnt, unvisited(), diff, before;
    @*/
  {
    while(stackPtr != 0) {
      int u = stack[--stackPtr];
      stack[stackPtr] = -1;

      if (ord[u] == -1) {
        ord[u] = cnt++;

        boolean[] adj = G[u];

        /*@ set diff = 0;
          @ set before = stackPtr;
          @*/

        /*@  requires \invariant_for(this);
          @  requires adj == G[u];
          @  requires 0 <= u && u < G.length;
          @  requires ord[u] != -1 && (cnt > \old(cnt) && ord[v] == \old(cnt));
          @  requires diff == 0 && stackPtr <= (cnt * G.length);
          @  requires stackPtr == before;
          @  ensures \invariant_for(this)
                && (cnt > \old(cnt) && ord[v] == \old(cnt));
          @  ensures diff <= ord.length - cnt;
          @  ensures stackPtr == before + diff;
          @ signals_only \nothing;
          @ assignable stack[stackPtr..stack.length], stackPtr, t, diff;
          @*/
```

```
          {
            /*@ loop_invariant \invariant_for(this);
              @ loop_invariant adj == G[u];
              @ loop_invariant 0 <= t && t <= adj.length;
              @ loop_invariant 0 <= u && u < G.length;
              @ loop_invariant ord[u] != -1;
              @ loop_invariant diff <= ord.length - cnt
                          && stackPtr < (cnt + 1) * G.length;
              @ loop_invariant stackPtr == before + diff;
              @ loop_invariant (\num_of int i; 0 <= i && i < t;
                                ord[i] == -1 && adj[i]) == diff;
              @
              @ decreases adj.length - t;
              @ assignable stack[stackPtr..stack.length], stackPtr, t, diff;
              @*/
            for(int t = 0; t < adj.length; t++) {
              if (adj[t] && ord[t] == -1) {
                /*@ set diff = diff + 1;
                  @*/
                stack[stackPtr++] = t;
              }
            }
          }

        }
      }
    }

  }

}
```

Listing A.11: Non-recursive DFT using an adjacency matrix

```
public class GraphDFS_adjarr_nonrec {
  /*@ public invariant (\forall int j; 0 <= j && j < len;
                        0 <= G[j] && G[j] < adjArr.length && (G[j] <= G[j+1]));
    @ public invariant len > 0;
    @ public invariant G != adjArr && adjArr != ord && ord != G
                           && G != stack && adjArr != stack;
    @ public invariant 0 <= G[len] && G[len] < adjArr.length;
    @*/
  int[] G;
```

```
/*@ public invariant (\forall int i; 0 <= i && i < adjArr.length;
                      0 <= adjArr[i] && adjArr[i] < len);
  @*/
int[] adjArr;

/*@ model int len;
  @ represents len \such_that len == G.length - 1;
  @*/

/*@ public invariant 0 <= cnt && cnt <= len;
  @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
  @*/
int cnt;

/*@ public invariant ord.length == len;
  @ public invariant (\forall int i; 0 <= i && i < cnt;
                      (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @*/
int[] ord;

/*@ public invariant 0 <= stackPtr && stackPtr <= stack.length
                           && stackPtr <= (cnt + 1)*len;
  @*/
int stackPtr;

/*@ public invariant stack.length == (len * len);
  @ public invariant (\forall int i; 0 <= i && i < stackPtr;
                      0 <= stack[i] && stack[i] < len);
  @ public invariant (\forall int i; stackPtr <= i && i < stack.length;
                      stack[i] == -1);
  @ public invariant stack != ord;
  @*/
int[] stack;
```

```
  /*@ public normal_behavior
    @ requires GIn != adjIn;
    @ requires (\forall int j; 0 <= j && j < GIn.length - 1;
                  0 <= GIn[j] && GIn[j] < adjIn.length && (GIn[j] <= GIn[j+1]));
    @ requires 0 <= GIn[GIn.length-1] && GIn[GIn.length-1] < adjIn.length;
    @ requires (\forall int i; 0 <= i && i < adjIn.length;
                  0 <= adjIn[i] && adjIn[i] < GIn.length-1);
    @ requires GIn.length - 1 > 0;
    @
    @ ensures G == GIn && adjArr == adjIn;
    @ ensures stackPtr == 0;
    @ ensures (cnt == 0);
    @ ensures (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
    @ ensures (\forall int i; 0 <= i && i < stack.length; stack[i] == -1);
    @*/
  public GraphDFS_adjarr_nonrec(int[] GIn, int[] adjIn) {
    G = GIn;
    adjArr = adjIn;
    cnt = 0;
    ord = new int[G.length-1];

    stackPtr = 0;
    stack = new int[(G.length - 1) * (G.length - 1)];

    /*@ loop_invariant 0 <= i && i <= ord.length;
      @ loop_invariant (\forall int j; 0 <= j && j < i; ord[j] == -1);
      @ decreases ord.length - i;
      @ assignable ord[i .. (ord.length - 1)], i;
      @*/
    for(int i = 0; i < ord.length; i++) {
      ord[i] = -1;
    }

    /*@ loop_invariant 0 <= k && k <= stack.length;
      @ loop_invariant (\forall int j; 0 <= j && j < k; stack[j] == -1);
      @ decreases stack.length - k;
      @ assignable stack[k .. (stack.length - 1)], k;
      @*/
    for(int k = 0; k < stack.length; k++) {
      stack[k] = -1;
    }
  }

  /*@ model \locset unvisited() {
          return \infinite_union(int i; (0 <= i && i < ord.length && ord[i] == -1);
              \singleton(ord[i]));
  } @*/
```

```
/*@ public normal_behavior
  @  requires cnt == 0 && stackPtr == 0;
  @  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
  @
  @  ensures (cnt == len);
  @  ensures (\forall int i; 0 <= i && i < len; ord[i] != -1);
  @  ensures (\forall int i; 0 <= i && i < len;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @ assignable ord[*], cnt, before, diff, stack[*], stackPtr;
  @*/
public void dft() {
  /*@ loop_invariant 0 <= k && k <= len;
    @ loop_invariant \invariant_for(this);
    @ loop_invariant (\forall int i; 0 <= i && i < k; ord[i] != -1);
    @ loop_invariant cnt >= k && stackPtr == 0;
    @ decreases len - k;
    @ assignable unvisited(), cnt, k, before, diff, stack[*], stackPtr;
    @*/
  for(int k = 0; k < G.length-1; k++) {
    if (ord[k] == -1){searchC(k);}
  }
}

/*@ ghost int diff;
  @ ghost int before;
  @*/
```

```
/*@ public normal_behavior
  @  requires cnt < len;
  @  requires 0 <= v && v < len;
  @  requires ord[v] == -1;
  @  requires stackPtr == 0;
  @
  @  ensures cnt > \old(cnt);
  @  ensures (ord[v] == \old(cnt)) && (ord[v] != -1);
  @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @  ensures stackPtr == 0;
  @ assignable unvisited(), cnt, stack[*], stackPtr, before, diff;
  @*/
```

```
public void searchC(int v) {
  stack[stackPtr++] = v;

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires 0 <= v && v < len;
    @  requires (stack[0] == v && ord[v] == -1 && stackPtr == 1 && cnt == \old(cnt))
           || (cnt > \old(cnt) && ord[v] == \old(cnt));
    @  requires 0 <= \old(cnt) && \old(cnt) <= len;
    @
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                  (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @  ensures \old(cnt) < cnt;
    @  ensures ord[v] == \old(cnt);
    @  ensures stackPtr == 0;
    @  ensures \invariant_for(this);
    @ decreases (len * len) - (cnt * len) + stackPtr;
    @ assignable stackPtr, stack[*], cnt, unvisited(), diff, before;
    @*/
  {
    while(stackPtr != 0) {
      int u = stack[--stackPtr];
      stack[stackPtr] = -1;

      if (ord[u] == -1) {
        ord[u] = cnt++;


        /*@ set diff = 0;
          @ set before = stackPtr;
          @*/



        /*@  requires \invariant_for(this);
          @  requires 0 <= u && u < len;
          @  requires ord[u] != -1 && (cnt > \old(cnt) && ord[v] == \old(cnt));
          @  requires diff == 0 && stackPtr <= (cnt * len);
          @  requires stackPtr == before;
          @
          @  ensures \invariant_for(this)
                 && (cnt > \old(cnt) && ord[v] == \old(cnt));
          @  ensures diff <= ord.length - cnt;
          @  ensures stackPtr == before + diff;
          @ signals_only \nothing;
          @ assignable stack[stackPtr..stack.length], stackPtr, t, diff;
          @*/
```

```
          {
            /*@ loop_invariant \invariant_for(this);
              @ loop_invariant G[u] <= t && t <= G[u+1];
              @ loop_invariant 0 <= u && u < len;
              @ loop_invariant ord[u] != -1;
              @ loop_invariant diff <= ord.length - cnt && stackPtr < (cnt + 1) * len;
              @ loop_invariant stackPtr == before + diff;
              @ loop_invariant (\num_of int i; G[u] <= i && i < t;
                                  ord[adjArr[i]] == -1) == diff;
              @
              @ decreases G[u+1] - t;
              @ assignable stack[stackPtr..stack.length], stackPtr, t, diff;
              @*/
            for(int t = G[u]; t < G[u+1]; t++) {
              if (ord[adjArr[t]] == -1) {
                /*@ set diff = diff + 1;
                @*/
                stack[stackPtr++] = adjArr[t];
              }
            }
          }

        }
      }
    }


  }

}
```

Listing A.12: Non-recursive DFT using an adjacency array

```
public class GraphDFS_link_nonrec {
  /*@ public invariant (\forall int j; 0 <= j && j < G.length;
                        G[j] != null && G[j].adj.length <= G.length);
    @ public invariant G.length > 0;
    @ public invariant (\forall int i; 0 <= i && i < G.length;
                         (\forall int j; 0 <= j && j < G[i].adj.length;
                             0 <= G[i].adj[j].val && G[i].adj[j].val < G.length
                          && G[i].adj[j] == G[G[i].adj[j].val]));
    @ public invariant (\forall int i; 0 <= i && i < G.length; G[i].val == i);
    @ public invariant (\forall int i; 0 <= i && i < G.length;
                         (\forall int j; 0 <= j && j < G.length;
                            (i!=j) ==> (G[i] != G[j]) ));
    @ public invariant (\forall int i; 0 <= i && i < G.length; \invariant_for(G[i]));
    @*/
  Vertex[] G;

  /*@ public invariant 0 <= cnt && cnt <= G.length;
    @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
    @*/
  int cnt;

  /*@ public invariant ord.length == G.length;
    @ public invariant (\forall int i; 0 <= i && i < cnt;
                         (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @*/
  int[] ord;

  /*@ public invariant 0 <= stackPtr && stackPtr <= stack.length && stackPtr <= (cnt +
      1)*G.length;
    @*/
  int stackPtr;

  /*@ public invariant stack.length == (G.length * G.length);
    @ public invariant (\forall int i; 0 <= i && i < stackPtr;
                         0 <= stack[i] && stack[i] < G.length);
    @ public invariant (\forall int i; stackPtr <= i && i < stack.length;
                         stack[i] == -1);
    @ public invariant stack != ord;
    @*/
  int[] stack;
```

```
/*@ public normal_behavior
  @ requires (\forall int j; 0 <= j && j < GIn.length;
                GIn[j] != null && GIn[j].adj.length <= GIn.length);
  @ requires (\forall int i; 0 <= i && i < GIn.length;
                (\forall int j; 0 <= j && j < GIn[i].adj.length;
                    0 <= GIn[i].adj[j].val && GIn[i].adj[j].val < GIn.length
                 && GIn[i].adj[j] == GIn[GIn[i].adj[j].val]));
  @ requires (\forall int i; 0 <= i && i < GIn.length; GIn[i].val == i);
  @ requires (\forall int i; 0 <= i && i < GIn.length;
                (\forall int j; 0 <= j && j < GIn.length; (i!=j) ==> (GIn[i] !=
                    GIn[j]) ));
  @ requires (\forall int i; 0 <= i && i < GIn.length; \invariant_for(GIn[i]));
  @ requires GIn.length > 0;
  @
  @ ensures G == GIn;
  @ ensures stackPtr == 0;
  @ ensures (cnt == 0);
  @ ensures (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
  @ ensures (\forall int i; 0 <= i && i < stack.length; stack[i] == -1);
  @*/
public GraphDFS_link_nonrec(Vertex[] GIn) {
  G = GIn;
  cnt = 0;
  ord = new int[G.length];

  stackPtr = 0;
  stack = new int[G.length * G.length];

  /*@ loop_invariant 0 <= i && i <= ord.length;
    @ loop_invariant (\forall int j; 0 <= j && j < i; ord[j] == -1);
    @ decreases ord.length - i;
    @ assignable ord[i .. (ord.length - 1)], i;
    @*/
  for(int i = 0; i < ord.length; i++) {
    ord[i] = -1;
  }

  /*@ loop_invariant 0 <= k && k <= stack.length;
    @ loop_invariant (\forall int j; 0 <= j && j < k; stack[j] == -1);
    @ decreases stack.length - k;
    @ assignable stack[k .. (stack.length - 1)], k;
    @*/
  for(int k = 0; k < stack.length; k++) {
    stack[k] = -1;
  }
}
```

```
/*@ model \locset unvisited() {
        return \infinite_union(int i; (0 <= i && i < ord.length && ord[i] == -1);
            \singleton(ord[i]));
} @*/


/*@ public normal_behavior
  @  requires cnt == 0 && stackPtr == 0;
  @  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
  @
  @  ensures (cnt == G.length);
  @  ensures (\forall int i; 0 <= i && i < G.length; ord[i] != -1);
  @  ensures (\forall int i; 0 <= i && i < G.length;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @ assignable ord[*], cnt, before, diff, stack[*], stackPtr;
  @*/
public void dft() {

  /*@ loop_invariant 0 <= k && k <= G.length;
    @ loop_invariant \invariant_for(this);
    @ loop_invariant (\forall int i; 0 <= i && i < k; ord[i] != -1);
    @ loop_invariant cnt >= k && stackPtr == 0;
    @ decreases G.length - k;
    @ assignable unvisited(), cnt, k, before, diff, stack[*], stackPtr;
    @*/
  for(int k = 0; k < G.length; k++) {
    if (ord[k] == -1){searchC(k);}
  }
}

/*@ ghost int diff;
  @ ghost int before;
  @*/

/*@ public normal_behavior
  @  requires cnt < G.length;
  @  requires 0 <= v && v < G.length;
  @  requires ord[v] == -1;
  @  requires stackPtr == 0;
  @
  @  ensures cnt > \old(cnt);
  @  ensures (ord[v] == \old(cnt)) && (ord[v] != -1);
  @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @  ensures stackPtr == 0;
  @ assignable unvisited(), cnt, stack[*], stackPtr, before, diff;
  @*/
```

```
public void searchC(int v) {
  stack[stackPtr++] = v;

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires 0 <= v && v < G.length;
    @  requires (stack[0] == v && ord[v] == -1 && stackPtr == 1 && cnt == \old(cnt))
           || (cnt > \old(cnt) && ord[v] == \old(cnt));
    @  requires cnt <= G.length;
    @  requires 0 <= \old(cnt) && \old(cnt) <= G.length;
    @
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @  ensures \old(cnt) < cnt;
    @  ensures ord[v] == \old(cnt);
    @  ensures stackPtr == 0;
    @  ensures \invariant_for(this);
    @ decreases (G.length * G.length) - (cnt * G.length) + stackPtr;
    @ assignable stackPtr, stack[*], cnt, unvisited(), diff, before;
    @*/
  {
    while(stackPtr != 0) {
      int u = stack[--stackPtr];
      stack[stackPtr] = -1;

      if (ord[u] == -1) {
        ord[u] = cnt++;

        Vertex[] adj = G[u].adj;

        /*@ set diff = 0;
          @ set before = stackPtr;
          @*/

        /*@  requires \invariant_for(this);
          @  requires adj == G[u].adj;
          @  requires 0 <= u && u < G.length;
          @  requires ord[u] != -1
                 && (cnt > \old(cnt) && ord[v] == \old(cnt));
          @  requires diff == 0 && stackPtr <= (cnt * G.length);
          @  requires stackPtr == before;
          @  ensures \invariant_for(this)&&(cnt > \old(cnt) && ord[v]==\old(cnt));
          @  ensures diff <= ord.length - cnt;
          @  ensures stackPtr == before + diff;
          @ signals_only \nothing;
          @ assignable stack[stackPtr..stack.length], stackPtr, t, diff;
          @*/
```

```
          {
            /*@ loop_invariant \invariant_for(this);
              @ loop_invariant adj == G[u].adj;
              @ loop_invariant 0 <= t && t <= adj.length;
              @ loop_invariant 0 <= u && u < G.length;
              @ loop_invariant ord[u] != -1;
              @ loop_invariant diff <= ord.length - cnt && stackPtr < (cnt + 1) *
                  G.length;
              @ loop_invariant stackPtr == before + diff;
              @ loop_invariant (\num_of int i; 0 <= i && i < t;
                                  ord[adj[i].val] == -1) == diff;
              @
              @ decreases adj.length - t;
              @ assignable stack[stackPtr..stack.length], stackPtr, t, diff;
              @*/
            for(int t = 0; t < adj.length; t++) {
              if (ord[adj[t].val] == -1) {
                /*@ set diff = diff + 1;
                  @*/
                stack[stackPtr++] = adj[t].val;
              }
            }
          }

        }

      }
    }


  }

}
```

Listing A.13: Non-recursive DFT using a linked data structure

```
public class GraphDFS_list_nonrec {
  /*@ public invariant (\forall int j; 0 <= j && j < G.length;
                        G[j] != null && G[j].size() <= G.length);
    @ public invariant G.length > 0;
    @ public invariant (\forall int i; 0 <= i && i < G.length;
                         (\forall int j; 0 <= j && j < G[i].size();
                             0 <= G[i].get(j) && G[i].get(j) < G.length));
    @ public invariant (\forall int i; 0 <= i && i < G.length; \invariant_for(G[i]));
    @ public invariant (\forall int i; 0 <= i && i < G.length;
                         \disjoint(G[i].footprint,\singleton(cnt))
                       && \disjoint(G[i].footprint, \singleton(stackPtr))
                       && (\forall int j; 0 <= j && j < ord.length;
                             \disjoint(G[i].footprint,\singleton(ord[j])))
                       && (\forall int l; 0 <= l && l < stack.length;
                             \disjoint(G[i].footprint,\singleton(stack[l]))) );
    @*/
  List[] G;

  /*@ public invariant 0 <= cnt && cnt <= G.length;
    @ public invariant (\num_of int i; 0 <= i && i < ord.length; ord[i] != -1) == cnt;
    @*/
  int cnt;

  /*@ public invariant ord.length == G.length;
    @ public invariant (\forall int i; 0 <= i && i < cnt;
                         (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @*/
  int[] ord;

  /*@ public invariant 0 <= stackPtr && stackPtr <= stack.length&& stackPtr <= (cnt +
      1)*G.length;
    @*/
  int stackPtr;

  /*@ public invariant stack.length == (G.length * G.length);
    @ public invariant (\forall int i; 0 <= i && i < stackPtr;
                         0 <= stack[i] && stack[i] < G.length);
    @ public invariant (\forall int i; stackPtr <= i && i < stack.length;
                         stack[i] == -1);
    @ public invariant stack != ord;
    @*/
  int[] stack;
```

```
  /*@ public normal_behavior
    @ requires (\forall int j; 0 <= j && j < GIn.length;
                  GIn[j] != null && GIn[j].size() <= GIn.length);
    @ requires (\forall int i; 0 <= i && i < GIn.length;
                  (\forall int j; 0 <= j && j < GIn[i].size();
                      0 <= GIn[i].get(j) && GIn[i].get(j) < GIn.length));
    @ requires (\forall int i; 0 <= i && i < GIn.length; \invariant_for(GIn[i]));
    @ requires GIn.length > 0;
    @
    @ ensures G == GIn;
    @ ensures stackPtr == 0;
    @ ensures (cnt == 0);
    @ ensures (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
    @ ensures (\forall int i; 0 <= i && i < stack.length; stack[i] == -1);
    @*/
  public GraphDFS_list_nonrec(List[] GIn) {
    G = GIn;
    cnt = 0;
    ord = new int[G.length];

    stackPtr = 0;
    stack = new int[G.length * G.length];

    /*@ loop_invariant 0 <= i && i <= ord.length;
      @ loop_invariant (\forall int j; 0 <= j && j < i; ord[j] == -1);
      @ decreases ord.length - i;
      @ assignable ord[i .. (ord.length - 1)], i;
      @*/
    for(int i = 0; i < ord.length; i++) {
      ord[i] = -1;
    }

    /*@ loop_invariant 0 <= k && k <= stack.length;
      @ loop_invariant (\forall int j; 0 <= j && j < k; stack[j] == -1);
      @ decreases stack.length - k;
      @ assignable stack[k .. (stack.length - 1)], k;
      @*/
    for(int k = 0; k < stack.length; k++) {
      stack[k] = -1;
    }
  }


  /*@ model \locset unvisited() {
          return \infinite_union(int i; (0 <= i && i < ord.length && ord[i] == -1);
              \singleton(ord[i]));
  } @*/
```

```
/*@ public normal_behavior
  @  requires cnt == 0 && stackPtr == 0;
  @  requires (\forall int i; 0 <= i && i < ord.length; ord[i] == -1);
  @
  @  ensures (cnt == G.length);
  @  ensures (\forall int i; 0 <= i && i < G.length; ord[i] != -1);
  @  ensures (\forall int i; 0 <= i && i < G.length;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @ assignable ord[*], cnt, before, diff, stack[*], stackPtr;
  @*/
public void dfs() {

  /*@ loop_invariant 0 <= k && k <= G.length;
    @ loop_invariant \invariant_for(this);
    @ loop_invariant (\forall int i; 0 <= i && i < k; ord[i] != -1);
    @ loop_invariant cnt >= k && stackPtr == 0;
    @ decreases G.length - k;
    @ assignable unvisited(), cnt, k, before, diff, stack[*], stackPtr;
    @*/
  for(int k = 0; k < G.length; k++) {
    if (ord[k] == -1){searchC(k);}
  }
}



/*@ ghost int diff;
  @ ghost int before;
  @*/



/*@ public normal_behavior
  @  requires cnt < G.length;
  @  requires 0 <= v && v < G.length;
  @  requires ord[v] == -1;
  @  requires stackPtr == 0;
  @
  @  ensures cnt > \old(cnt);
  @  ensures (ord[v] == \old(cnt)) && (ord[v] != -1);
  @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
  @  ensures stackPtr == 0;
  @ assignable unvisited(), cnt, stack[*], stackPtr, before, diff;
  @*/
```

```
public void searchC(int v) {
  stack[stackPtr++] = v;

  /*@ loop_contract normal_behavior
    @  requires \invariant_for(this);
    @  requires 0 <= v && v < G.length;
    @  requires (stack[0] == v && ord[v] == -1 && stackPtr == 1 && cnt == \old(cnt))
           || (cnt > \old(cnt) && ord[v] == \old(cnt));
    @  requires 0 <= \old(cnt) && \old(cnt) <= G.length;
    @
    @  ensures (\forall int i; \old(cnt) <= i && i < cnt;
                  (\exists int j; 0 <= j && j < ord.length; ord[j] == i));
    @  ensures \old(cnt) < cnt;
    @  ensures ord[v] == \old(cnt);
    @  ensures stackPtr == 0;
    @  ensures \invariant_for(this);
    @ decreases (G.length * G.length) - (cnt * G.length) + stackPtr;
    @ assignable stackPtr, stack[*], cnt, unvisited(), diff, before;
    @*/
  {
    while(stackPtr != 0) {
      int u = stack[--stackPtr];
      stack[stackPtr] = -1;

      if (ord[u] == -1) {
        ord[u] = cnt++;

        List adj = G[u];

        /*@ set diff = 0;
          @ set before = stackPtr;
          @*/

        /*@  requires \invariant_for(this);
          @  requires 0 <= u && u < G.length && adj == G[u];
          @  requires ord[u] != -1
                 && (cnt > \old(cnt) && ord[v] == \old(cnt));
          @  requires diff == 0 && stackPtr <= (cnt * G.length);
          @  requires stackPtr == before;
          @
          @  ensures \invariant_for(this)
                && (cnt > \old(cnt) && ord[v] == \old(cnt));
          @  ensures diff <= ord.length - cnt;
          @  ensures stackPtr == before + diff;
          @ signals_only \nothing;
          @ assignable stack[stackPtr..stack.length], stackPtr, t, diff;
          @*/
```

```
          {
            /*@ loop_invariant \invariant_for(this);
              @ loop_invariant 0 <= t && t < adj.size();
              @ loop_invariant 0 <= u && u < G.length && adj == G[u];
              @ loop_invariant ord[u] != -1;
              @ loop_invariant diff <= ord.length - cnt
                          && stackPtr < (cnt + 1) * G.length;
              @ loop_invariant stackPtr == before + diff;
              @ loop_invariant (\num_of int i; 0 <= i && i < t;
                                ord[adj.get(i)] == -1) == diff;
              @
              @ decreases adj.size() - t;
              @ assignable stack[stackPtr..stack.length], stackPtr, t, diff;
              @*/
            for(int t = 0; t < adj.size(); t++) {
              if (ord[adj.get(t)] == -1) {
                /*@ set diff = diff + 1;
                  @*/
                stack[stackPtr++] = adj.get(t);
              }
            }
          }

        }
      }
    }


  }

}
```

Listing A.14: Non-recursive DFT using adjacency lists

```
import java.text.SimpleDateFormat;
import java.util.Date;

void setup() {
  String path = sketchPath();

  JSONObject log = loadJSONObject(path + "/data/log_time.txt");

  Table perFile = new Table();
  perFile.addColumn("date");
  perFile.addColumn("day");
  perFile.addColumn("time");
  perFile.addColumn("body");
  perFile.addColumn("file");

  SimpleDateFormat date = new SimpleDateFormat("yyyy-mm-dd hh:mm:ss X");
  SimpleDateFormat day = new SimpleDateFormat("yyyy-mm-dd");
  SimpleDateFormat time = new SimpleDateFormat("hh:mm:ss");

  JSONArray commits = log.getJSONArray("commits");

  for (int i = 0; i < commits.size();i++) {
    println(i + "/" + commits.size());
    JSONObject commit = commits.getJSONObject(i);

    String dateStr = commit.getString("date");
    String body = commit.getString("body");
    String changeStr = commit.getString("change");
    Date d = new Date();

    try {d = date.parse(dateStr);} catch (Exception e) {}

    String[] files = parseChangedFiles(changeStr);
    TableRow row;

    for(String file : files) {
      row = perFile.addRow();
      row.setString("date",dateStr);
      row.setString("day",day.format(d));
      row.setString("time",time.format(d));
      row.setString("body",body);
      row.setString("file",file);
    }
  }
```

```
  println("saving");
  saveTable(raw, path + "/data/raw_output_aug.csv");
  saveTable(perCommit, path + "/data/commit_changes_aug.csv");
  saveTable(perFile, path + "/data/file_changes_aug.csv");
  println("done");
}

String[] parseChangedFiles(String changes) {
  String[] lines = changes.split("\n");

  ArrayList<String> list = new ArrayList<String>();

  for(String s : lines) {
    if(match(s,"\\|") != null) {
      //parse changed file
      String[] parts = s.split("\\|");

      for (int i = 0; i < parts.length - 1; i++) {
        String[] sub = parts[i].split("\\+");
        sub = sub[sub.length-1].split("-");
        list.add(sub[sub.length-1]);
      }

    }

  }

  String[] res = new String[list.size()];
  for(int i = 0; i < list.size(); i++) {
    res[i] = list.get(i);
  }

  return res;
}
```

Listing A.15: Processing-script to parse json-commits to csv