Answering reachability queries on large directed graphs

Introducing a new data structure using bit vector compression

Author:
Sebastiaan J. van Schaik
Utrecht University / University of Oxford

Supervisors:
Prof. Oege de Moor (University of Oxford)
Prof. dr. Arno Siebes (Utrecht University)

Universiteit Utrecht

Thesis submitted in partial fulfilment of the degree of Master of Science in Computer Science
Department of Computing Sciences, Faculty of Science, Utrecht University

September 2010

Answering reachability queries on large directed graphs

Introducing a new data structure using bit vector compression

Sebastiaan J. van Schaik
Utrecht University / University of Oxford

Abstract

Answering reachability queries on graphs has been subject of extensive research for the last couple of decades. Reachability data structures and algorithms provide an answer to probably one of the easiest sounding questions in graph theory: can some vertex j be reached from another vertex i along edges in the graph? Such queries can be answered in many different ways, for example by using data structures representing the transitive closure of a graph.

Starting in the 1950's, computer scientists and mathematicians have proposed multiple ways to process a graph, extract reachability information and represent the transitive closure. Data sources – and the graphs representing them – are vastly growing, forcing researchers to look for new ways to efficiently represent a transitive closure, which grows quadratically in the number of vertices of a graph. Additionally, the amount of time required to process a graph and build a transitive closure data structure should be limited, as well as the amount of time required to answer reachability queries using the data structure.

This thesis provides an introduction to transitive closure computation and proposes to use the concept of bit vector compression to reduce the amount of memory required to represent a transitive closure. A new data structure (based on bit vector compression) is presented, together with both a theoretical and experimental analysis to compare its performance – in terms of memory usage, construction time and query response time – to data structures presented in publications at major conferences.

Although data structures described in recent publications are supported by a very sound theoretical foundation, it turns out that in practice more trivial existing approaches to compression of transitive closure data structures often provide similar or even better performance. The newly designed compression scheme often works faster (in terms of both construction time and query time) and has a smaller memory footprint in virtually all cases.

"Life is all about timing... the unreachable becomes reachable, the unavailable become available, the unattainable... attainable. Have the patience, wait it out. It is all about timing."
— Stacey Charter

Contents

1 Introduction **1**
- 1.1 Introduction . . . 1
- 1.2 Preliminary definitions . . . 2
- 1.3 Transitive closure . . . 3
- 1.4 Contributions . . . 5
- 1.5 Structure of the thesis . . . 5

2 Prior work **7**
- 2.1 Floyd-Warshall . . . 7
- 2.2 Exploiting strongly connected components . . . 8
- 2.3 Using chain and path decomposition . . . 16
- 2.4 Matrix multiplication . . . 20

3 Compressing reachability information **21**
- 3.1 Introduction . . . 21
- 3.2 WAH . . . 21
- 3.3 PWAH . . . 23

4 Experimental evaluation **33**
- 4.1 Set up of experiments . . . 33
- 4.2 Influence of input graphs on PWAH performance . . . 36
- 4.3 Comparing different PWAH schemes . . . 41
- 4.4 Indexing PWAH . . . 42
- 4.5 PWAH vs. interval lists, Path-Tree and 3-Hop . . . 45

5 Conclusion **53**
- 5.1 Experimental evaluation . . . 53
- 5.2 Further research . . . 53

A Tarjan's algorithm: example **57**

B Code details **61**

Chapter 1

Introduction

1.1 Introduction

1.1.1 Reachability

Providing reachability information about a graph within a limited amount of time using a limited amount of memory is still a challenging problem in computer science. Although the problem itself almost seems too trivial to be interesting, it has been a subject of ongoing research for the last couple of decades. Extensive work has been done not only to decrease the amount of time required to answer a reachability query, but also to optimise (in terms of both time and memory usage) the process of building a data structure required to answer such queries.

1.1.2 Applications

Reachability queries are used in many kinds of applications, both in research and in industry. Examples of such applications include, but are certainly not limited to:

Source code analysis – a call graph provides information about method calls. Reachability information of such a graph can help to find *dead code*: methods which can and will never be reached from an application's *main* method. Other examples of source code analysis using reachability information are *pointer analysis* [3] and interprocedural dataflow analysis [26, 27].

Analysis of social networks – social networks can be modelled as graphs capturing people (vertices) and connections between people (edges). Reachability information can be used to determine whether a person is a friend of a friend of $\cdots$ of a friend, enabling the network to pose some restrictions on whether or not a person can view another person's profile page.

Computational biology – when analysing *metabolic networks* [20], reachability information is used to assist in solving search problems. Furthermore, reachability information has been used for classification of cardiac abnormalities by using heart rate signals [1] and for querying gene ontologies represented by graphs [4, 9, 35].

Model checking – reachability analysis is a very important part of model checking [5]: it helps determining whether a state (or set of states) can reach certain other states. States are represented by vertices in a graph, state changes are modelled by edges. These graphs can easily grow extremely large, making it harder and harder to determine from which states an unwanted state can be reached.

Route planning – when travelling from some location a to an other location b, people are mostly interested in computing the shortest (or fastest) path and knowing the exact route. On first sight, it does not seem to be very interesting to know *whether or not* a path exist. Hence, reachability

information does not seem to suffice. However, it might be interesting be able to determine quickly whether a path without using e.g. a ferry or a toll road exists. Would reachability information be available, portable navigation devices can instantly inform the user that it is indeed not possible to travel from e.g. Edinburgh to Lerwick on the Shetland islands without taking a ferry, in stead of having to analyse a very large number of alternative routes in order to be able to present that conclusion.

1.2 Preliminary definitions

Answering queries regarding reachability within graphs is strongly related to *transitive closure* computation. Before describing the concept of *transitive closure*, it is important to introduce some graph theoretic notation of basic concepts. Notation varies in literature, the notation and concepts stated below will be used throughout this thesis.

Directed graph

Definition 1. *A* directed graph *(or* digraph*) G is a pair (V, E), where V is a finite set of size n and E is a binary relation on V. The set $V = \{v_0, v_1, \ldots, v_{n-2}, v_{n-1}\}$ is called the* vertex set *of G, and its elements are called* vertices*. The set E is called the* edge set *of G, and its elements are called* edges*. An edge is an ordered pair of vertices (v_i, v_j) which is referred to by, respectively, the* tail *and the* head *of the edge.*
(adapted from [7, 34])

Although research in the field of symmetrical reachability (on *undirected* graphs, in which edges consist of *unordered* pairs of vertices) is closely related to reachability in *directed* graphs, the algorithms and data structures discussed in this thesis are devised to process *directed* graphs only. Therefore, when referring to a graph, the qualifier *'directed'* is implied unless stated otherwise.

Subgraph

Definition 2. *A graph $G' = (V', E')$ is a* subgraph *of $G = (V, E)$ if and only if $V' \subseteq V$ and $E' \subseteq E$. Given a set $V' \subseteq V$, the subgraph of G induced by V' is the graph $G' = (V', E')$ where $E' = \{(v_i, v_j) \in E : v_i, v_j \in V'\}$.*
(from [7])

Path and path length

Definition 3. *A* path *from vertex v_0 to vertex v_k in G, denoted $v_0 \xrightarrow{*} v_k$, is a (possibly empty) sequence of edges of the form $\langle (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \rangle$, where each edge is in E. The* length *of a path is the number of edges in it. A path from v_0 to v_k is* non-null*, denoted $v_0 \xrightarrow{+} v_k$ if its length is positive. A path is* simple *if all edges and vertices on the path, except possibly the first and the last vertex, are distinct.*
(adapted from [24])

Cycle, acyclic

Definition 4. *A* cycle *is a non-null simple path that begins and ends at the same vertex. A graph that contains no cycles is* acyclic*.*
(from [24])

For most algorithms and data structures described in this thesis, graphs are not required to be acyclic. Therefore, unless stated otherwise, it is assumed graphs may contain cycles.

Adjacency matrix

Definition 5. *An* adjacency matrix *M of a graph $G = (V, E)$ is a $|V| \times |V|$ Boolean matrix such that $M[i, j]$ is true if and only if $(v_i, v_j) \in E$ $(v_i, v_j \in V)$.*
(adapted from [24])

Adjacency matrices provide an $\mathcal{O}(1)$ lookup time to determine the existence of an edge, but generally require $\mathcal{O}(|V|^2)$ memory. Enumerating all edges is an $\mathcal{O}(|V|^2)$ time operation. For large graphs it is not feasible to store adjacency information using an adjacency matrix. For example: provided that the adjacency matrix is allowed to take up to 512 MB of memory, the graph can consist of at most 64,000 vertices.

Adjacency list

Definition 6. *An* adjacency list *L of vertex $v \in V$ is a list that contains the vertices adjacent to v. The adjacency list representation of a graph G consists of the adjacency lists of its vertices.*
(from [24])

In contrast to an adjacency matrix, adjacency lists of a graph G use only $\mathcal{O}(|E|)$ memory in total. For most graphs, $|E| \ll |V| \cdot |V|$ and therefore, adjacency lists generally take less memory to represent a graph. On the other hand, the list does not provide an $\mathcal{O}(1)$ query time. Provided the list is sorted, the query time is $\mathcal{O}(log_2(|V|))$. For an unsorted list, the query time increases to $\mathcal{O}(|V|)$. However, enumerating all edges in a graph takes only $\mathcal{O}(|E|)$ time.

Topological sort

Definition 7. *A* topological sort *of a directed acyclic graph $G = (V, E)$ is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.*
(from [7])

Note that a cyclic graph $H = (V, E)$ can never have a consistent topological sort, since it is impossible to impose a linear ordering on all vertices in a cycle such that u appears before v whenever an edge (u, v) exists. Also note that a graph can have more than one valid topological ordering.

1.3 Transitive closure

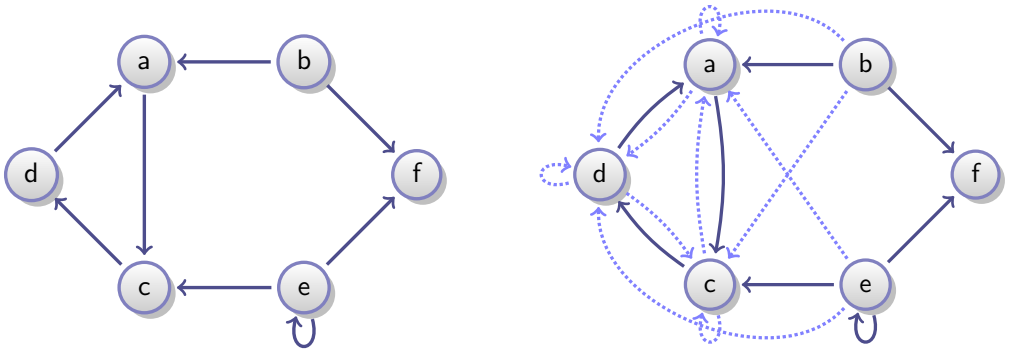
1.3.1 Introduction and definition

The transitive closure of a graph $G = (V, E)$ is defined as a derived graph $G^+ = (V, E^+)$, in which E^+ denotes the smallest transitive relation that contains E. In other words: the transitive closure G^+ will explicitly contain an edge (v_i, v_j) if and only if a path $v_i \xrightarrow{+} v_j$ exists in G. The process of computing the *transitive closure* of a graph can be thought of as creating a data structure which represents reachability information about a graph G.

Figure 1.1 on the following page depicts an example graph G and its transitive closure G^+. The light blue dotted edges represent the edges which were added:

$$E^+ - E = \{(a, a), (a, d), (b, c), (b, d), (c, a), (c, c), (d, c), (d, d), (e, a), (e, d)\}$$

Table 1.1 on the next page shows the adjacency lists of both graphs. By definition (and as is clearly illustrated in the figure and can be deduced from the adjacency lists), the edge set E^+ of a transitive closure G is a superset of the original edge set E of graph G: $E \subseteq E^+$. The vertex set V is equal for both G and G^+.



(a) Graph G

(b) Transitive closure G^+ of G

Figure 1.1: Original graph G and transitive closure G^+

v	adjacent vertices
a	c
b	a, f
c	d
d	a
e	c, e, f
f	

(a) Adjacency lists of vertices in graph G

v	adjacent vertices
a	$\mathbf{a}, c, \mathbf{d}$
b	$a, \mathbf{c}, \mathbf{d}, f$
c	$\mathbf{a}, \mathbf{c}, d$
d	$a, \mathbf{c}, \mathbf{d}$
e	$\mathbf{a}, c, \mathbf{d}, e, f$
f	

(b) Adjacency lists of vertices in G^+

Table 1.1: Adjacency lists of vertices from graphs G and G^+

1.3.2 Reflexive transitive closure

The *reflexive* transitive closure $G^* = (V, E^*)$ is closely related to the 'regular' transitive closure. The set of edges E^+ (of the regular transitive closure) contains an edge (v_i, v_j) if and only if a path $v_i \xrightarrow{+} v_j$ exists in G, i.e. null paths are excluded. The reflexive transitive closure does not exclude null paths, hence E^* contains an edge (v_i, v_j) if and only if a path $v_i \xrightarrow{*} v_j$ exists in G. Note that $E^+ \subseteq E^*$. These additional edges in E^* do not introduce any significant additional complexity to the process of computing the transitive closure.

Some publications (including [18, 19], described in Section 2.3) and books refer to the *reflexive* transitive closure simply as *transitive closure*. The research described in this thesis primarily deals with *non-reflexive transitive closures*. Unless mentioned otherwise, reflexivity is *not* implied.

1.3.3 Types of queries

Different problems ask for different types of reachability information. In general, the following types of queries can be identified:

- *Single pair* reachability queries:
  given two vertices $v_i, v_j \in V$, does there exist an edge $(v_i, v_j) \in E^+$?
- *Single source* reachability queries:
  given a vertex $v_i \in V$, enumerate all vertices $v_j \in V$ for which $(v_i, v_j) \in E^+$;
- *Multi-source* reachability queries:
  given a set of vertices $W \subseteq V$, which vertices are reachable from W?

When certain query characteristics are known beforehand, trivial pruning techniques can be applied (as described in Section 2.2.6 on page 16) to decrease the amount of time and memory

required to compute and store the data structures. However, this thesis primarily focuses on data structures which are capable of answering any type of query for each and every vertex (or pair of vertices) in a graph. In order to be able to do so, a representation of the full transitive closure needs to be computed and stored.

1.3.4 Challenges

As will be shown in Chapter 2, it is not particularly hard to extract reachability information from a graph. Challenges primarily lie within finding ways to *store* the transitive closure of a graph using only a limited amount of memory.

In general, two extreme approaches can be identified:

- *Use no memory for data structure, spend a lot of time on queries*
  This approach does not involve any preprocessing of the graph and typically uses $\mathcal{O}(|V| + |E|)$ time per query source by performing a DFS/BFS *on demand*. Clearly, this approach is infeasible when a large amount of queries needs to be processed.
- *Use a lot of memory, answer queries very fast*
  By preprocessing the graph and storing reachability information using an $|V| \times |V|$ matrix, query time can be highly optimised. The most extreme approach (using a reachability matrix) typically uses $\mathcal{O}(|V|^2)$ memory and provides an $\mathcal{O}(1)$ time performance for a single-pair reachability query.

1.4 Contributions

The main contributions of this thesis can be summarised as follows:

- The usage of bit vector compression is suggested, in order to reduce the amount of memory required to store reachability information;
- A new fine-grained run-length compression scheme for bit vectors is introduced, suitable for compressing reachability information;
- The performance of the new compression scheme is evaluated by theoretical analysis and by experiments on graphs;
- Using both randomly generated and real-world graphs, multiple approaches to computing and storing a transitive closure are compared.

1.5 Structure of the thesis

After having introduced both the concept of reachability and the definition of transitive closure in this chapter, this thesis will focus on describing prior work in Chapter 2. This ranges from a short description of the Floyd-Warshall algorithm (1959 – 1962, [12,29]) to compute the transitive closure of a graph, to more recent publications on computing and efficiently storing a transitive closure.

In Chapter 3, a new way of storing reachability information by means of bit vector compression is introduced. This includes a description of an existing bit compression scheme, but mainly focuses on the introduction of a newly designed scheme.

Chapter 4 describes a large number of experiments carried out using a number of different data structures and algorithms from Chapters 2 and 3. Graphs of varying sizes and from different sources are processed, including randomly generated graphs. The different approaches are compared in terms of memory usage, construction time and query time.

Finally, all theoretical and experimental results are wrapped up in Chapter 5. This chapter also poses some open questions and describes some suggestions for future research.

Chapter 2

Prior work

2.1 Floyd-Warshall

2.1.1 Introduction

The Floyd-Warshall algorithm for all-pairs shortest path computation was described independently by both Robert Floyd [12] and Stephen Warshall [40] in 1962. Roughly the same algorithm was published three years earlier by Bernard Roy [29], therefore literature sometimes refers to the Floyd-Warshall algorithm as the Roy-Floyd algorithm or Roy-Floyd-Warshall. The total running time of the algorithm is $\Theta(|V|^3)$, whilst using $\Theta(|V|^2)$ memory. Although these figures render this approach unusable for processing large graphs, the algorithm is briefly described in this section for it is considered to be the first algorithm capable of extracting reachability information from a graph.

The algorithm is a textbook example of *dynamic programming* and computes the shortest path (which can be infinite, or better: non-existing) between every pair of vertices in a graph within $\Theta(|V|^3)$ time. An edge length can be assigned to every edge, but this additional information is irrelevant when only the transitive closure is of interest. Therefore, edge lengths are fixed to 1 in this short overview of the algorithm.

2.1.2 The algorithm

The algorithm is best explained by expressing it as a recursive function. Let `shortPath(k,i,j)` denote the shortest path from vertex $v_i \in V$ to $v_j \in V$, using only vertices $v_1, \ldots, v_k \in V$.

Algorithm 1 Floyd-Warshall's algorithm

$$shortPath(0,i,j) = \begin{cases} \infty & if\ (i,j) \notin V \\ 1 & if\ (i,j) \in V \end{cases}$$

$$\begin{aligned} &shortPath(k,i,j) = min(\\ &\quad shortestPath(k-1,i,j), \\ &\quad shortestPath(k-1,i,k) + shortestPath(k-1,k,j) \\ &) \end{aligned}$$

The function `shortPath` is called for every pair (v_i, v_j) starting with $k = 0$, storing the path lengths in a *memoisation table* (sometimes referred to as *dynamic programming table*). Note that at time k is still 0, most path lengths are probably infinite. After each iteration, k is increased by 1 and more paths will emerge. When $k = n$, all paths have been analysed and the algorithm has finished.

The process is based on the following two important observations:

1. $k = 0$ is the recursion base: it defines the case in which a path from v_i to v_j is not allowed to be routed via any other intermediate vertex. Obviously, such a path exists if and only if an edge (v_i, v_j) exists in edge set E.
2. When increasing k to $k + 1$, two situations might occur:
   (a) The shortest path only uses intermediate vertices in $\{v_1, \ldots, v_k\}$, the addition of v_{k+1} to the set of allowed intermediate vertices did not yield a new shorter path;
   (b) A shorter path emerges, using intermediate vertex v_{k+1}. This path consists of two parts: a sub path from v_i to v_{k+1} and a sub path from v_{k+1} to v_j.

After the last pass, the algorithm has produced the memoisation table for $k = n$. In other words: all vertices in the graph have been taken into consideration. Using this table, not only the shortest path from any pair of vertices (v_i, v_j) is known, but it is also possible to reconstruct this path using regular dynamic programming techniques, which lie beyond the scope of this thesis.

2.1.3 Analysis

The most significant advantage of the Floyd-Warshall algorithm over the other algorithms and data structures presented in this thesis, is that it provides the length of the shortest paths together with a way of actually reconstructing the paths. However, when answering reachability queries, the path length is not of any interest, nor is the exact routing of the path. Especially when dealing with very large graphs, the algorithm would yield an extremely long processing time and use significantly more memory when compared to the other approaches. Therefore, the Roy-Floyd-Warshall algorithm has not been included in the experimental analysis in this thesis.

2.2 Exploiting strongly connected components

2.2.1 Introduction

Computer scientists and mathematicians have endeavoured to minimise the processing time of a directed graph and the amount of memory required to store the transitive closure. Avoiding performing redundant computations is considered to be one of the most important challenges in constructing a transitive closure. Especially the generation of duplicate edges (edges which do not introduce any additional reachability information) turns out to be a major waste of time and memory. It has been shown [13] that the majority of duplicate edges is caused by the existence of *strongly connected components*. Hence, finding and merging these components can help in further optimising processing time and memory usage.

2.2.2 Strongly connected components

A strongly connected component consists of one or more vertices which are all fully connected to each other. Therefore, all vertices in the component can reach a common set of vertices outside the component (i.e., the vertices share the same reachability information).

The formal definition of a strongly connected component follows from the definition of the concepts *strongly connected* and *strongly connected subgraph*. All three definitions are included below:

Definition 8. *Two vertices $v_i, v_j \in V$ are* strongly connected *if and only if there exists a path $v_i \xrightarrow{*} v_j$ and a path $v_j \xrightarrow{*} v_i$.*

Definition 9. *A* strongly connected subgraph *$G' = (V', E')$ of G is a subgraph (see Definition 2 on page 2) of G induced by V' in which every pair of vertices $v_i, v_j \in V'$ is strongly connected.*

Definition 10. *A* maximal strongly connected subgraph *or* strongly connected component *$G' = (V', E')$ of G is a strongly connected subgraph of G, which can not be extended by adding one or more additional vertices v_k from V to V' whilst preserving the property of G' being a strongly connected subgraph of G.*

It is important to observe that every vertex $v_i \in V$ belongs to exactly one strongly connected component. Furthermore, would every strongly connected component be contracted to a single vertex, the resulting graph (called *condensed graph*, *condensation graph* or simply *condensation*) is acyclic. Detailed proofs of these observations are provided in [34].

Following from Definitions 8 to 10 on the facing page and the relevant observations, it is very likely that computation of reachability information of a graph G can be optimised (both in terms of memory usage and construction time) by using the condensation of G instead of G itself.

2.2.3 Tarjan's algorithm

Introduction

Before introducing one of the most widely referenced algorithms for constructing a transitive closure, it is important to have seen the algorithm on which it is based: Tarjan's algorithm. The procedure described by Robert Tarjan [34] in 1972 efficiently (within $\mathcal{O}(|V| + |E|)$ time) discovers strongly connected components of a graph by conducting a Depth First Search (see [7], pages 540-549) on a graph. Although this thesis refers to 'Tarjan's algorithm'. the procedure described in the paragraphs below incorporates improvements suggested by Esko Nuutila in his PhD thesis [24].

The algorithm – textual description

The algorithm uses a series of bookkeeping data structures to detect strongly connected components in a graph $G = (V, E)$:

- An array of components C to store the *final component* for each vertex $v \in V$;
- A stack of vertices S. The stack will store vertices which were visited by the DFS procedure, but which were not yet assigned a final component;
- An array of integers D to store the DFS sequence number of each vertex. Vertices will be assigned a sequence number in topological order;
- An array of vertices R to store the most suitable candidate component root (see below) for each vertex $v \in V$.

The most important notion of Tarjan's algorithm is that of a *candidate component root* (or *CCR*). Since no complete knowledge of the topology of the graph is available at the time of traversing the graph, it is most certainly not possible to identify a component once having seen its first vertex. Hence, every vertex which might be the first member of a new component is referred to as the CCR. Moreover, every candidate component is identified by its CCR. Note that during graph traversal, the algorithm might discover that two distinct candidate components should be merged into a single component. In that case, the value of D (DFS sequence number) of both CCR vertices is compared. The CCR with the largest value of D will be discarded in favour of the other CCR. In other words: the most recently detected candidate component is discarded in favour of the vertex which was discovered first.

Once a the DFS call has been returned to a vertex v_i by one of its children v_j, v_i should check:

1. whether v_j has obtained knowledge about a more suitable CCR, i.e.: $D[CCR[v_j]] < D[CCR[v_i]]$, and
2. whether $CCR[v_j]$ is a *valid* CCR for v_i (see below).

The value $CCR[v_j]$ is valid for v_i if and only if v_i and v_j belong to the same component. The two vertices belong to the same component if and only if $C[v_j]$ (representing the *final* component of v_j) is still undefined. If $CCR[v_j] < CCR[v_i]$ and $C[v_j] = \textit{undefined}$, the value $CCR[v_i]$ should be replaced: $CCR[v_i] := CCR[v_j]$.

After having processed all of its children, v_i has obtained sufficient knowledge about the relevant graph topology to be able to decide whether or not v_i is a *final* component root. If none of the children has provided v_i with a more suitable CCR (i.e., $CCR[v_i]$ has not been changed, v_i is still its own CCR: $CCR[v_i] = v_i$), v_i will create a new component c and pop vertices $W \subseteq V$ off stack S until $Top(S) = v_i$. These vertices belong to the new component c, therefore: $\forall v_j \in W : C[v_j] := c$.

For a more technical description and proofs of correctness, please refer to the original paper "Depth-first search and linear graph algorithms" by Robert Tarjan [34], as well as [24].

The algorithm – pseudocode

The following pseudocode shows the details of Tarjan's DFS procedure:

Algorithm 2 Tarjan's algorithm

```
1:  procedure DfsVisit(v)
2:      CCR[v] ← v                          ▷ initially, v is its own CCR
3:      C[v] ← Nil                          ▷ the final component of v is undefined
4:      D[v] ← d++                          ▷ store DFS sequence number of v
5:      for all w : (v, w) ∈ E do           ▷ loop over all children of v
6:          if w not visited then DfsVisit(w)   ▷ call DFS on child w (if not visited yet)
7:          if C[w] = Nil and D[CCR[w]] < D[CCR[v]] then
8:              CCR[v] ← CCR[w]             ▷ update CCR of v if necessary
9:          end if
10:     end for
11:     if CCR[v] = v then                  ▷ after DFS, check if v is still its own CCR
12:         C[v] ← new component
13:         while D[Top(S)] > D[v] do       ▷ pop vertices off the stack
14:             w ← Pop(S)
15:             C[w] ← C[v]
16:         end while
17:     else
18:         Push(v, S)                      ▷ push v on stack for later processing
19:     end if
20: end procedure
21: procedure main(G = (V, E))
22:     for all v ∈ V do
23:         if v not visited then DfsVisit(v)   ▷ initiate DFS on vertex v
24:     end for
25: end procedure
```

Example

The following step-by-step example and the graph depicted in Figure 2.1a on the next page are adapted from [24]. Note that the example provided below only contains the steps processing the first 6 vertices. The full list of steps can be found in Appendix A on page 57.

1. DfsVisit(a) – Visit vertex a:
   (a) Mark a as CCR of itself: $CCR[a] = a$, store DFS sequence number 1 for a: $D[a] = 1$
   (b) Iterate over adjacent vertices: DfsVisit(b)

2. DfsVisit(b) – Visit vertex b:
   (a) Mark b as CCR of itself: $CCR[b] = b$, store DFS sequence number 2 for b: $D[b] = 2$
   (b) Iterate over adjacent vertices:
      i. Adjacent vertex a has been visited before, not calling DfsVisit
      $C[a] = Nil$ and $D[CCR[a]] < D[CCR[b]]$ $(1 < 2)$,
      therefore b inherits the CCR of a: $CCR[b] \leftarrow CCR[a]$ (thus, $CCR[b] = a$)
      ii. Adjacent vertex c has not been visited before: DfsVisit(c)

3. DfsVisit(c) – Visit vertex c:
   (a) $CCR[c] = c$, $D[c] = 3$

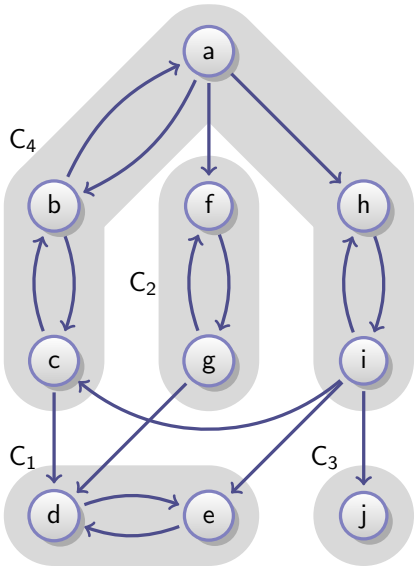
(b) Iterate over adjacent vertices:
   i. Adjacent vertex b has been visited before, not calling DfsVisit
   $C[b] = Nil$ and $D[CCR[b]] < D[CCR[c]]$ $(1 < 3)$,
   therefore c inherits the CCR of b: $CCR[c] \leftarrow CCR[b]$ (thus, $CCR[c] = a$)
   ii. Adjacent vertex d has not been visited before: DfsVisit(d)

4. DfsVisit(d) – Visit vertex d:
   (a) $CCR[d] = d$, $D[d] = 4$
   (b) Iterate over adjacent vertices: DfsVisit(e)

5. DfsVisit(e) – Visit vertex e:
   (a) $CCR[e] = e$, $D[e] = 5$
   (b) Iterate over adjacent vertices:
      i. Adjacent vertex d has been visited before, not calling DfsVisit
      $C[d] = Nil$ and $D[CCR[d]] < D[CCR[e]]$ $(4 < 5)$,
      therefore e inherits the CCR of d: $CCR[e] \leftarrow CCR[d]$ (thus, $CCR[e] = d$)
   (c) No more adjacent vertices. $CCR[e] = d(\neq e)$, therefore push e on stack S: $S = \{e\}$
   (d) Return DFS call to d

6. Continue processing at vertex d
   (c) No more adjacent vertices left. $CCR[d] = d$, therefore d becomes *final* component root
      i. Create a new component: $C[d] = new\ component$
      ii. Stack $S = \{e\}$, pop component vertices off the stack:
      Pop $e \rightarrow C[e] = C[d]$
      $S = \{\}$
   (d) Return DFS call to c

7. Continue processing at vertex c
   (c) No more adjacent vertices left. $CCR[c] = a(\neq c)$, therefore push c on stack S: $S = \{c\}$



(a) Graph G

Vertex	Adjacency list	Comp.
a	$\{b, f, h\}$	C_4
b	$\{a, c\}$	C_4
c	$\{b, d\}$	C_4
d	$\{e\}$	C_1
e	$\{d\}$	C_1
f	$\{g\}$	C_2
g	$\{d, f\}$	C_2
h	$\{i\}$	C_4
i	$\{c, e, h, j\}$	C_4
j	$\{\}$	C_3

(b) Adjacency lists of G

Figure 2.1: Example graph (adapted from [24]) consisting of 10 vertices residing within 4 strongly connected components

(d) Return DFS call to b

8. ...

(continued in Appendix A on page 57)

Reverse topological order

At this point, it is important to notice that Tarjan's algorithm will process the strongly connected components in a *reverse* topological order (see Definition 7 on page 3). Suppose C_l is reachable from C_i via some path $p = \{(C_i, C_j), \cdots, (C_k, C_l)\}$, then Tarjan's algorithm will detect C_l prior to detecting C_i. Intuitively, this means that the algorithm will travel as 'deep' as possible (after all, it is conducting a *depth* first search) and start wrapping up strongly connected components on its way back. This property will turn out to be very useful when processing reachability information.

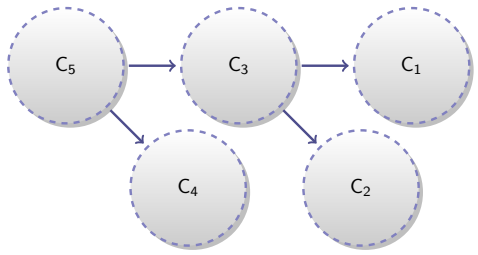
2.2.4 Nuutila's algorithm

Introduction: merging reachability information

Tarjan's algorithm is one of many ways to detect strongly connected components, but the procedure itself does not yield a result which directly provides us with reachability information. Of course it is possible to conduct another depth first search pass on the condensed graph resulting from Tarjan's algorithm to extract the wanted information. However, in 1995, Nuutila [24] introduced a significant extension to Tarjan's algorithm which extracts reachability information from an input graph without requiring an additional pass over the graph. In this thesis, the procedure STACK_TC described by Nuutila in [24] will be referred to as Nuutila's algorithm.

Recall that Tarjan's algorithm detects the strongly connected components (or simply 'components') of the graph in reverse topological order. Hence, the component which is to be detected first (denoted C_0) can never reach any other component. In general: for $i < j$, a component C_j can never occur in the *successor set* of a component C_i.

Furthermore – again because of the order in which the components are detected – it is clear that the successor set of a component C_i can be constructed by merging the successor sets of its adjacent components. A very simple illustration of the process of merging reachability information is shown in Figure 2.2.



(a) A condensation graph G'

Comp.	Reachability information
C_1	$R(C_1) = \varnothing$
C_2	$R(C_2) = \varnothing$
C_3	$R(C_3) = R(C_1) \cup R(C_2) \cup \{C_1, C_2\}$
C_4	$R(C_4) = \varnothing$
C_5	$R(C_5) = R(C_3) \cup R(C_4) \cup \{C_3, C_4\}$

(b) Reachability information

Figure 2.2: Merging reachability information $R(C_i)$ for components in a condensed graph G'

Algorithm description

The crux of Nuutila's algorithm lies within the detection of edges between components *while* traversing the graph to detect the components. Edges between components are generally referred to as *inter-component edges* – e.g. (a, f), (c, d), (g, d), (i, e) and (i, j) in the example graph depicted in Figure 2.3 on the facing page.

Nuutila's algorithm employs an extra stack to store these inter-component edges at the moment they are encountered during graph traversal. At the time a new component is constructed by its

final component root, it is possible to iterate over the adjacent components by popping the correct number of components from the stack. In the same pass, reachability information of all adjacent components can be merged to construct the aggregated reachability information which is to be stored in the newly created component.

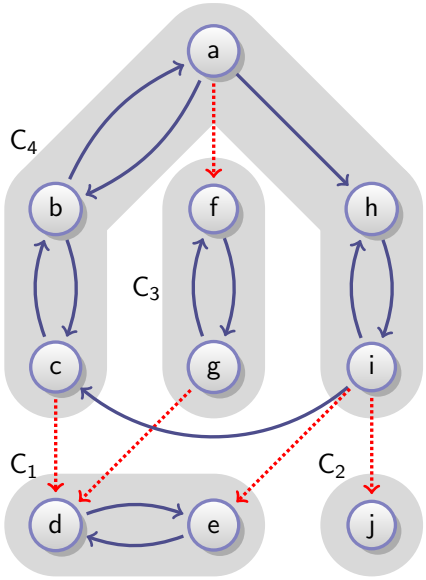


Figure 2.3: Inter-component edges (dotted red) in the example graph

Pseudocode

Nuutila's algorithm uses four additional data structures for bookkeeping purposes:

- A stack of components S_C. Note that the vertex stack S as described in Section 2.2.3 on page 9 will be called S_V from now on;
- An array of integers H to keep track of the height of stack S_C during graph traversal;
- An array of booleans L to record self-loops of vertices;
- A vector of reachability data structures R for strongly connected components.

The algorithm can use various types of data structures (e.g. a lists of integers or reachability bit vectors) to store the actual reachability information of strongly connected components. Section 2.2.5 on the following page contains an extensive description of a number of different types of data structures.

Algorithm 3 on page 15 lists the pseudocode for Nuutila's algorithm. An extensive step-by-step walkthrough using the running example graph is omitted, since the algorithm does very much look like Tarjan's algorithm of which a step-by-step description is included in Appendix A. However, there are some parts of the algorithm which might need further explanation:

l. 17 A *forward edge* is an edge (v_i, v_j) whilst a non-null path $v_i \xrightarrow{+} v_j$ was already discovered earlier. A forward edge does not introduce any additional knowledge of the graph topology (regarding reachability information), and can therefore be ignored.

l. 24 In a *regular* transitive closure (as opposed to a *reflexive* transitive closure) an edge $(v, v) \in E^+$ only exists if and only if a non-null path $v \xrightarrow{+} v$ exists in the original graph. When using a condensed graph to optimise transitive closure computation, this property is to be preserved. Therefore, it is important to realise that there exist two distinct cases in which a strongly component C_i can reach itself:

1. C_i consists of only a single vertex v *and* a self-loop $(v, v) \in E \implies C_i$ can reach itself;
2. C_i consists of > 1 vertices $\implies C_i$ can reach itself.

A strongly connected component consisting of only a single vertex is sometimes called a *trivial strongly connected component*. Would the one and only vertex v of a trivial component C_i not have a self-loop (i.e. $(v, v) \notin E$), then C_i should not appear in its own successor set. Nuutila's algorithm will initialise the reachability information (successor set) of a newly created component $C[v]$ on line 25 or 27, depending on whether the component is supposed to appear in its own successor set or not.

l. 32 On this line, the algorithm checks whether an adjacent component X already appears in the successor set of the newly created component $C[v]$, in order to prevent merging the same successor sets twice. This check highly depends on the lookup performance of the successor set implementation. In case a lookup of X in $C[v]$ would yield higher costs than conducting a possibly duplicate merge operation, it is imperative that the check should be removed.

Run-time analysis

Just like Tarjan's, the performance of Nuutila's algorithm highly depends on the topology of the input graph. Furthermore, the implementation of the successor set data structure is of utmost importance. Nuutila provides a very detailed asymptotic run-time analysis in his thesis, in which is shown that the algorithm runs in $\mathcal{O}(|V| \cdot |E| + |V|^2)$ time when using simple data structures like bit vectors or integer lists. Section 2.2.5 provides an overview of popular data structures to represent successor sets.

2.2.5 Representing successor sets

Algorithms for constructing a transitive closure highly depend on the performance of the underlying data structure used to represent successor sets. In particular, algorithms like Nuutila's depend on the time it takes to merge two or more of these successor sets. Although almost all data structures will perform that operation in $\mathcal{O}(|V|)$ time, the actual time it takes varies quite significantly as is shown in Chapter 4 on page 33.

This section briefly describes a number of fairly well known data structures to represent successor sets, including their (dis)advantages. As is often the case in computer science, choosing a data structure boils down to making a trade-off between time and memory.

Bit vectors

The most significant advantage of using bit vectors to represent successor sets, is the speed with which computers can merge bit vectors. Basically, the merge operation boils down to performing a logical OR on a series of bit vectors. Another advantage which is most certainly worth mentioning, is the $\mathcal{O}(1)$ lookup time of a random bit in the bit vector.

Unfortunately, bit vectors can grow extremely large, since it is required to store a `0`-bit for every non-existing edge as well. From that perspective, bit vectors grow linearly to the number of vertices $|V|$ of a graph, rather than the number of edges $|E|$ as most other data structures do. Storing all bit vectors will therefore use $\mathcal{O}(|V|^2)$ memory.

Lists of integers

Although storing a successor sets as a list of integers would use an amount of memory which will grow linearly in the number of edges (rather than the number of vertices), representing a single edge will typically require 32 bits. The lookup time is $\mathcal{O}(log_2(|V|))$ for sorted lists and $\mathcal{O}(|V|)$ for unsorted lists.

Algorithm 3 Nuutila's algorithm

```
1: procedure NUUTILA(v)
2:     CCR[v] ← v                                  ▷ initially, v is its own CCR
3:     C[v] ← Nil                                  ▷ the final component of v is undefined
4:     D[v] ← d++                                  ▷ store DFS sequence number of v
5:     Push(v, S_V)                                ▷ push v on vertex stack
6:     H[v] ← HEIGHT(S_C)                          ▷ store height of component stack S_C
7:     L[v] ← false                                ▷ possible self-loop of v still to be detected
8:     for all w : (v, w) ∈ E do                   ▷ loop over all children of v
9:         if w = v then
10:            L[v] ← true                         ▷ record self-loop of v
11:        else
12:            if w not visited then DFSVISIT(w)   ▷ call DFS on child w
13:            if C[w] = Nil then
14:                if D[CCR[w]] < D[CCR[v]] then
15:                    CCR[v] ← CCR[w]             ▷ update CCR of v
16:                end if
17:            else if (v, w) is not a forward edge then
18:                PUSH(C[w], S_C)                 ▷ inter-component edge
19:            end if
20:        end if
21:    end for
22:    if CCR[v] = v then                          ▷ after DFS, v is still its own CCR
23:        C[v] ← new component
24:        if TOP(S_V) ≠ v or L[v] then
25:            R[C[v]] ← {C[v]}                    ▷ initialise reachability info: self-loop
26:        else
27:            R[C[v]] ← ∅                         ▷ initialise reachability info: none
28:        end if
29:
30:        while HEIGHT(S_C) ≠ H[v] do             ▷ process adjacent components
31:            X ← POP(S_C)
32:            if X ∉ R[C[v]] then                 ▷ prevent performing duplicate operations
33:                R[C[v]] ← R[C[v]] ∪ R[X] ∪ {X}  ▷ merge reachability information
34:            end if
35:        end while
36:        repeat                                  ▷ pop the vertex stack S_V
37:            w ← POP(S_V)
38:            C[w] ← C[v]
39:        until w = v
40:    end if
41: end procedure
42: procedure MAIN(G = (V, E))
43:    for all v ∈ V do
44:        if v not visited then DFSVISIT(v)       ▷ initiate DFS on vertex v
45:    end for
46: end procedure
```

Interval lists

Interval lists are very similar to lists of integers, but are capable of storing contiguous sequences of integers much more efficiently by representing the sequence by a pair consisting of the first and the last integer. For example, the integer list $\{2, 3, 4, 5, 8, 9, 10, 11\}$ can be compressed by representing it as an interval list: $\{(2, 5), (8, 11)\}$. Although interval lists are very suitable for storing a contiguous sequence of integers, using them for storing non-contiguous sequences will not decrease memory usage. It is shown in [24] that the maximum number of intervals required to store a sequence of integers $S \subseteq \{1, 2, \ldots, n\}$ equals $\lceil \frac{n}{2} \rceil$.

Recall that components of a graph are created in a reverse topological order by the DFS process. It is therefore very likely that the integer list representation will consists of sequences of contiguous numbers. Figure 2.2 on page 12 might provide some intuition regarding this observation. Using interval lists to represent successor sets can therefore turn out to be very effective.

Assuming the interval list is sorted (which is a reasonable assumption, since the interval lists of components are created in reverse topological order), the query time is $\mathcal{O}(log_2(|L|)) \leq \mathcal{O}(log_2(|V|))$ (where $|L|$ denotes the number of intervals in the interval list).

2.2.6 Graph pruning

As stated before, this thesis primarily describes data structures and algorithms for computing and storing reachability information for all pairs of vertices in a graph. The procedure MAIN (starting one line 42 of Algorithm 3 on the previous page) therefore considers every single vertex v of graph $G = (V, E)$. However, if reachability information of only a limited subset of vertices $W \subseteq V$ is considered to be of interest, it is sufficient to call procedure NUUTILA using only vertices from W. Note that it is hard to determine beforehand whether this pruning technique will actually yield a better result, since a single call to the NUUTILA DFS procedure might process the entire graph anyway.

2.3 Using chain and path decomposition

2.3.1 Introduction

A popular technique of compressing a transitive closure representation is using a *path decomposition* or *chain decomposition* [14, 17–19, 30] of a graph. Note that there does not seem to exist a broadly agreed definition of a path decomposition of a directed graph.

2.3.2 Path-Tree

Introduction and criticism

The Path-Tree algorithm by Jin *et al.* [19] presented at the SIGMOD conference on management of data in 2008 introduces a new technique for computing and storing a representation of the transitive closure. This thesis will describe the approach described by Jin *et al.* in considerable detail, since it is one of the most recent and significant publications on this topic. Furthermore, it illustrates the use of path decomposition for compressing the transitive closure.

Jin *et al.* provide the following definition of a path decomposition:

> *"Let $G = (V, E)$ be a directed acyclic graph. We say a partition $P_1, \cdots, P_k$ of V is a path decomposition of G if and only if $P_1 \cup \cdots \cup P_k = V$ and $P_i \cap P_j = \varnothing$ for any $i \neq j$. We also refer to k as the width of the decomposition."*

However, this definition does not provide a solid definition of a path; it solely defines a partitioning of vertices. This partitioning is not bound by the structure of the graph (edges) and allows to divide the vertex set in groups of totally unconnected vertices. Although the authors do not present a proper definition of a path decomposition, Figure 2.4a on page 18 provides an illustration of the

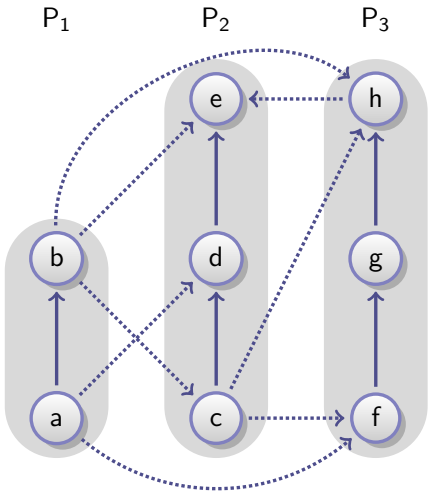
concept. A path is actually a sequence of vertices, rather than just a partition: each vertex has an unique position in the path, defined by the topological order of the graph.

Algorithm description

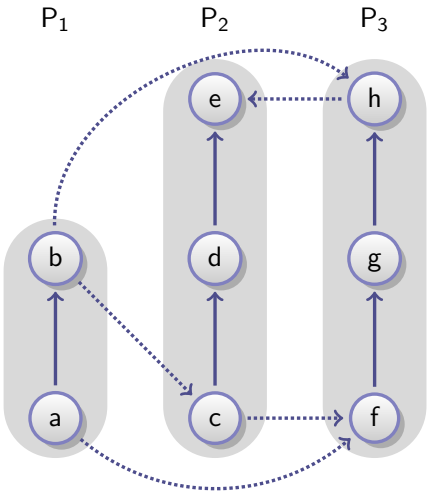
Jin *et al.* present a new data structure called *Path-Tree*, which consists of a tree shaped spanning subgraph of a directed acyclic graph. A labelling for the graph is devised, which allows efficient query processing. The approach consists of a number of steps, of which an extensive explanation and proofs can be found in [19]. An overview of these steps is provided below.

0. If the input graph $G' = (V', E')$ is not acyclic, apply Tarjan's algorithm to construct the condensed graph $G = (V, E)$ of G'. Otherwise, $G = G'$. This transformation is not considered to be a part of the algorithm, it is merely a preprocessing step.
1. Construct a path decomposition $P = P_1, \ldots, P_k$ of G. By definition, every vertex is part of exactly one path P_i and can therefore be identified by a tuple (i, s): i denoting the index of the path, s denoting the v's relative order (*sequence number* or *sid*) within the path. The vertex which is to appear as the first vertex in the topological ordering of the path is assigned sequence number 1. The authors introduce an additional notation:
   $$u \preceq v \iff v.sid \leq u.sid \text{ and } u, v \in P_i$$
   Denoting that u precedes (or equals) v in a path P_i. An example of a path decomposition is depicted in Figure 2.4a on the next page, the dotted edges are edges between paths (sometimes referred to as *inter-path edges*).
2. For each pair of paths P_i, P_j $(P_i \neq P_j)$, consider the set of edges $E_{P_i \to P_j} \subseteq E$ consisting of all edges of which the tail lies within P_i and the head in P_j (the dotted edges in Figure 2.4 on the following page). Construct the *minimal equivalent edge set* $E^R_{P_i \to P_j}$ by removing all edges from $E_{P_i \to P_j}$ which can be removed without affecting the reachability information of vertices in P_i with respect to vertices in P_j.

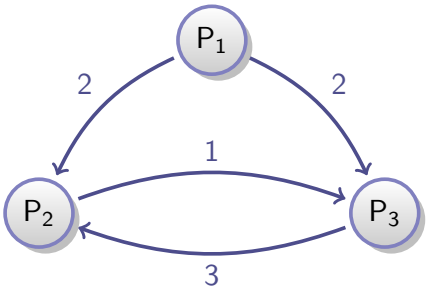
   An example of a minimal equivalent edge set of a path decomposition is depicted in Figure 2.4b on the next page. For example, the edge (b, e) is removed in the minimal equivalent edge set, since b can reach e by using the path $b \to c \to d \to e$. Note that constructing the minimal equivalent edge set for every pair of paths does not necessarily remove all redundant edges in a global sense. For example, edge (b, h) can be considered redundant, since b can reach h by $b \to c \to f \to g \to h$. However, this route consists of vertices from more than two paths, which is why the edge (b, h) is not considered redundant after processing all *pairs* of paths.
3. Construct the *path graph* $G_P = (V_P, E_P)$ based on the path decomposition of G by representing each path of vertices by a single vertex in G_P. The edges in the path graph represent the inter-path edges from the path decomposition, the weight of an edge $(P_i, P_j) \in E_P$ equals the number of vertices in path P_i which can reach vertices in P_j by using edges from the minimal equivalent edge set $E^R_{P_i \to P_j}$. For example, both $a, b \in P_1$ can reach P_2, yielding an edge weight of 2 for edge (P_1, P_2) in the path graph. The path graph of the example graph referred to earlier is shown in Figure 2.4c on the following page.
4. Construct the maximal directed spanning tree G^T_P of the path graph G_P, as depicted in Figure 2.4d on the next page.
5. Partition the maximal spanning tree G^T_P into separate paths. In the example graph depicted in Figure 2.4d, only one path can be constructed from the spanning tree: P_1, P_3, P_2. Based on each of these paths in G^T_P, a path-path graph G_{PP} can be constructed, like depicted in Figure 2.5a on page 19. The path-path graph contains the original vertices (from G) belonging to the paths $P_1, \ldots, P_n$ in the maximum spanning tree G^T_P, connected by the internal path edges (between vertices in the same path) and edges from the minimal equivalent edge set. The path-path graph is used to assign two labels X_i and Y_i to every vertex $v_i \in G_{PP}$:



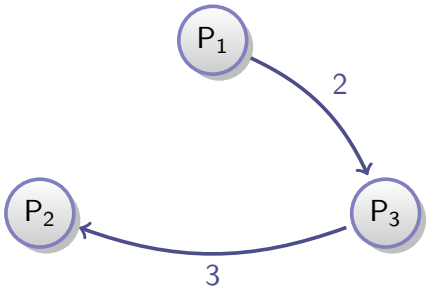
(a) A path decomposition $P = \{P_1, P_2, P_3\}$



(b) The minimal equivalent edge set of path in P



(c) path graph G_P

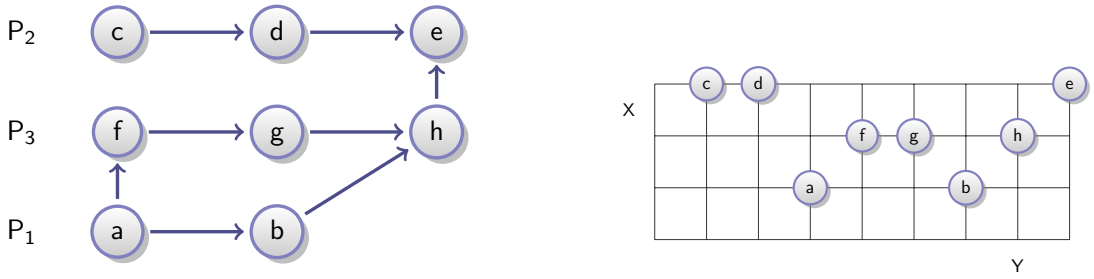


(d) The maximal directed spanning tree of G_P

Figure 2.4: Steps from a path decomposition to the maximal directed spanning tree of a path graph

- All vertices which are members of the same path P_i, will have an identical value of Y. The value of Y depends on the order in which the paths appear in the path-path graph. Therefore (based on Figure 2.4d):
  $a, b \in P_1 \longrightarrow Y = 1$
  $f, g, h \in P_3 \longrightarrow Y = 2$
  $c, d, e \in P_2 \longrightarrow Y = 3$
- The X value is assigned to a vertex $v_i \in G_{PP}$ by conducting a special depth-first search procedure, keeping track of a counter N (initially $N = |V|$). A vertex is assigned the value of N after all its adjacent vertices have been visited, after which the counter is decreased by one. The DFS starts at the first vertex of the first path of the path-path graph.
  In the example graph, vertex a would be visited first, being the first vertex of path P_1. The DFS procedure prioritises adjacent vertices from the same path over adjacent vertices from another path. Therefore, vertex b is the next vertex to be visited. From b only h can be visited, followed by e. Vertex e is the first vertex to be assigned a value for X: $X_e = N = 8$. Vertex h does not have any other adjacent vertices left to visit, therefore $X_h = 7$ and $X_b = 6$. Having returned at vertex a, there is a next adjacent vertex f to be visited: $X_g = 5, X_f = 4$. The DFS call started at a now ends at a, yielding $X_a = 3$. A new DFS procedure is started on the first non-visited vertex, iterating over the path in

the order specified by the maximal directed spanning tree. It starts at c, yielding $X_d = 2$ and finally $X_c = 1$. The values of X and Y which were assigned to the vertices are plotted in Figure 2.5b.



(a) The path-path graph G_{PP} of $P_1 \to P_3 \to P_2$

(b) The assigned values of X and Y

Figure 2.5: path-path graph and the assignment of values for X and Y

Based on the values of X and Y, the authors present the following lemma:

> *"Given two vertices u and v in the path-path graph, u can reach v if and only if $X_u \leq X_v$ and $Y_u \leq Y_v$ (this is also referred to as u dominates v)."*

Although this is true for the reachability within the path-path graph G_{PP} shown in Figure 2.5a, this lemma does *not* hold for the input graph G. For example, b is able to reach c in G even though $X_b \not\leq X_c$. This reachability information was lost after removing edges from the path graph G_P to construct its maximal spanning tree (Figure 2.4 on the facing page). To overcome this problem, the authors suggest to perform a regular on-demand search or to use an auxiliary data structure to answer queries related to vertices and edges not covered by their data structure.

To reduce the number of edges which are not covered by the data structure, the authors present the optimal path decomposition problem and map it to the minimum cost flow problem [16]. As is proved in [19] and also shown in Chapter 4, constructing an optimal path decomposition indeed improves the compression of the transitive closure representation.

Run-time analysis

Strangely, the authors do not provide a clear asymptotic run-time analysis to explain their claim [18] of providing an $\mathcal{O}(mk')$ / $\mathcal{O}(mn)$ (where $m = |E|, n = |V|, k' =$ number of paths in path decomposition of G) upper bound for constructing their data structure.

The authors' claim (section 2 of [19]) of providing an $\mathcal{O}(1)$ query time is most certainly not substantiated and rather misleading. It is true that this query time can be achieved for parts of the input graph, but the actual asymptotic time to answer a random query is $\mathcal{O}(log_2(k))$ (where k denotes the number of paths in the path decomposition). Chapter 4 contains an extensive experimental analysis of the query time, as well as a performance comparison to other approaches.

Note that the Path-Tree data structure is not capable of iterating over the transitive closure, it can only answer a reachability query regarding two distinct vertices. Tt is very hard to answer a *single* or *multi-source* reachability query using Path-Tree.

2.3.3 3-Hop

The 3-Hop approach was also presented at by Jin *et al.* at SIGMOD, one year after having introduced Path-Tree. A detailed description (like that of [19]) goes beyond the scope of this thesis, but 3-Hop can be considered an interesting approach to transitive closure computation. The crux of the algorithm lies within finding chains which can be seen as highways of the graph, as is depicted in Figure 2.6. A possible path from a vertex v to w will try to find a suitable highway (hop 1), take the appropriate exit (hop 2) and reach w (hop 3).

In their paper, the authors emphasise that 3-Hop is particularly suitable or processing directed acyclic graphs with a relatively large number of edges (a dense graph). This is illustrated in the paper by a very brief experimental analysis of query time using a limited data set. The authors do not present an experimental analysis of the construction time of their approach, nor do they present an extensive comparison to other approaches (like interval lists). Section 4.5.5 contains a more thorough experimental analysis of 3-Hop, including a comparison with other data structures and information regarding the construction time of 3-Hop.

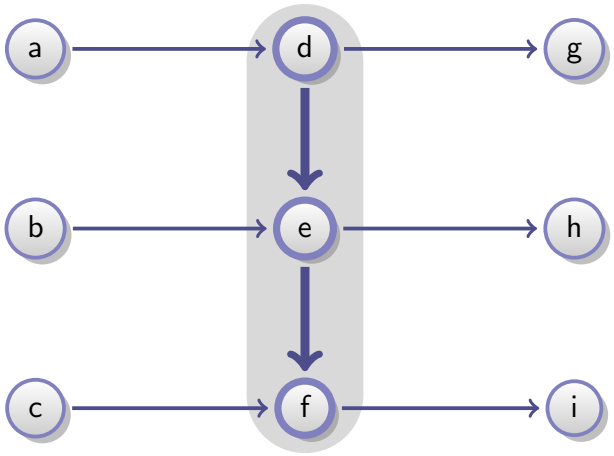


Figure 2.6: 3-Hop highway

2.4 Matrix multiplication

A very different approach to transitive closure computation, is matrix multiplication. As is shown in [11,21], the transitive closure of a graph G can be computed using a sequence of self-multiplications of the $|V| \times |V|$ Boolean adjacency matrix of G.

Although a naive approach to matrix multiplication would yield an $\mathcal{O}(n^3)$ algorithm, the Strassen [32] ($\mathcal{O}(n^{log_2(7)}) = \mathcal{O}(n^{2.807})$) and Coppersmith-Winograd [6] ($\mathcal{O}(n^{2.2376})$) algorithms provide a more interesting asymptotic upper bound. However, the enormous constant coefficient hidden within the Big-$\mathcal{O}$ of the Coppersmith-Winograd algorithm renders the algorithm only usable for graphs of sizes way beyond the graphs discussed in this thesis [28]. The algorithm described by Strassen – which also comes with a significant constant – is generally considered to be interesting for matrices with dimensions > 100 [25].

The construction of the transitive closure adjacency matrix would require a sequence of self-multiplications of the adjacency matrix of a graph G. However, it has been proved [11,21] that this sequence of multiplications can be carried out in a highly optimised way. Using this approach – the description of which lies far beyond the scope of this thesis – the time required to compute the transitive closure equals the time which is required to perform one Boolean matrix multiplication: $\mathcal{O}(n^{2.2376})$ using the Coppersmith-Winograd algorithm.

Although the approach of matrix multiplication is a very interesting one, overcoming the practical problems in an efficient implementation would be a separate research project in itself, which we defer to future research. Therefore, matrix multiplication has not been implemented and will not be evaluated in Chapter 4.

Chapter 3

Compressing reachability information

3.1 Introduction

As has been pointed out before, the reverse topological order in which the graph's strongly connected components are detected, yields highly clustered reachability information. Hence, the information can be efficiently represented as interval lists, as described in Chapter 2.

The same chapter also contains a description of the bit vector data structure, pointing out its main advantages (fast merging of successor lists, $\mathcal{O}(1)$ lookup time) and disadvantage (memory usage). Keeping in mind that reachability information is highly clustered, it is expected that storing a successor list as a bit vector will yield large sequences of contiguous `0`-bits and `1`-bits. This property makes the bit vectors extremely suitable for compression.

Another very important property of the reverse topological order in which the components are detected is the order in which the subsequent successor lists are constructed. As explained in Section 2.2.4 on page 12 and depicted in figure 2.2, a component C_j will never occur in the successor list of a component C_i ($i < j$, i and j part of a reverse topological ordering). This property opens the door to *run-length encoding* (RLE): computing and compressing reachability information at the same time.

This chapter suggests the usage of bit vector compression for fast and memory-efficient transitive closure computation. Although the concept of bit vector compression is not innovative, using it for storing reachability information has not been described in literature before.

There exist many run-length encoding techniques for bit vector compression, which are used in various fields of research. For example, compressed bit vectors are used in the field of data base research to compress bit map indices [43] and in the area of energy physics [31]. This chapter describes an existing technique proposed by Wu *et al.* in [42]. Furthermore, it introduces a novel compression scheme with some additional features, derived from the scheme described by Wu *et al.*.

3.2 Word Aligned Hybrid Compression

3.2.1 Introduction

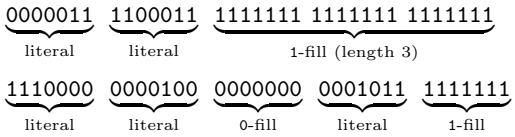
In [42], Wu *et al.* introduce a new run-length compression scheme called *Word Aligned Hybrid Compression* or simply *WAH*. The scheme processes a stream of uncompressed bits word by word and stores sequences of bits using a *fill word* or a *literal word*.

A *fill word* is used to compress a contiguous sequence of `1`-bits or `0`-bits and simply stores the length of the sequence in multiples of the block size (see below).

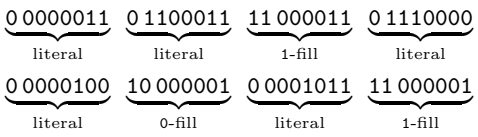
A *literal word* is used whenever a sequence of bits is not contiguous for a sufficient number of bits, and can therefore not be stored using a fill word. A literal word – as the term suggests – is a literal copy of the uncompressed bits.

A word is assumed to be of size 32 bits ($w = 32$), but might consist if any arbitrary number of bits (like $w = 64$, as is used in modern `x64` computers). The first (most significant) bit of a word in a WAH compressed bit vector indicates the type of the word: a `1`-bit denotes a fill, a `0`-bit indicates a literal word. In a fill word, the second most significant bit is used to denote the fill type (either a `0`-fill or a `1`-fill). The remaining $w - 2$ bits are used to store the fill length expressed in multiples of $w - 1$, which is referred to as the *block size*: $b = w - 1$. In a literal word, all b bits are used to store one block of uncompressed bits. Figure 3.1 shows an example bit vector and its WAH compressed representation, using a word size of $w = 8$ bits (instead of 32) and a block size $b = w - 1 = 7$ (instead of 31) for illustrational purposes.

Note that the length of a fill is expressed in multiples of the block size, seven in the example figure. In other words: the compression scheme *aligns* the words. Although it would have been possible to express the length of a fill in single bits (which would obviously yield a 'tighter' scheme and a better compression), it would result in significantly higher costs of processing. Wu *et al.* show in [42] that their scheme combines the best of two worlds and achieves a significant compression with low processing costs.



(a) 70 uncompressed bits in 10 blocks of 7 bits



(b) compressed representation of bit vector using 64 bits (8 words of 8 bits)

Figure 3.1: Example of a WAH compressed bit vector

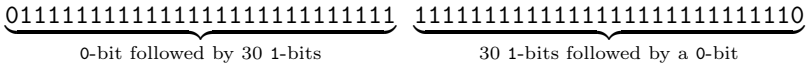
3.2.2 Basic operations

It is clear that the WAH scheme is a run-length encoding scheme: it processes the input bits word by word, it is always possible to set or clear a bit at the end of the compressed bit vector. At the same time it is not possible to set or clear bits which were already compressed, since that might result in having to split a single fill word into multiple pieces (or the other way around). However, when processing the components of the graph in reverse topological order, there is no need to set or clear random bits in a bit vector, as long as constructing a new bit vector based on two or more other bit vectors (by logical OR) is possible.

Computing a logical OR based on two input bit vectors can be done quite efficiently. In fact, the operation can be performed at least as fast – and most of the time even faster – than the same operation on corresponding uncompressed bit vectors, since a large amount of consecutive identical bits can be processed more efficiently. For transitive closure computation (merging successor lists), there is no need for any other operation than setting bits and merging bit vectors.

3.2.3 Limitations of WAH compression

The WAH compression scheme combines speed with significant compression, but will only yield a good result when the input is highly clustered. However, because of the fixed block size, the scheme will fail when the input does not contain sequences of `1`-bits of `0`-bits of sufficient length. Take a look at the following example ($w = 32, b = 31$):



Although the bit vector appears to be highly clustered, WAH can do nothing but create two literal words to represent the bits. This results in an overhead of one bit for every 31 bits, which equals about 3.2%. The example shows that WAH might end up storing 60 consecutive 1-bits using uncompressed literal words.

Furthermore, WAH allocates $w - 2$ bits to express the length of a fill. When dealing with a word size of 32 bits, 30 bits are allocated to express the fill length, yielding a maximum fill length of $2^{30} - 1 = 1,073,741,823$ blocks (33,285,996,513 bits). For a word size of 64 bits ($b = 63$), the maximum fill length grows to $2^{62} - 1 \approx 4.61 \times 10^{18}$. Most graphs do not have a number of vertices even close to that amount and therefore, fill lengths will consists of mostly zeroes resulting in a pure waste of memory.

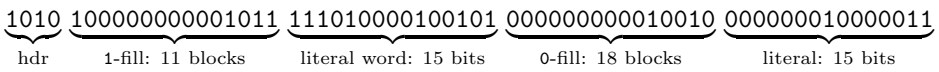
A smaller block size would result in a more fine grained compression scheme and would solve both shortcomings described above, which is why we designed *Partitioned Word Aligned Hybrid Compression* (or PWAH).

3.3 Partitioned Word Aligned Hybrid Compression

3.3.1 Introduction

This section introduces a novel compression scheme: Partitioned Word Aligned Hybrid compression – PWAH. It very much resembles the WAH [42] compression scheme, but adds the concept of a *partition*: a part of a word which can represent either a fill or a literal. Using partitions, the block size can be reduced, yielding a significantly more fine-grained compression scheme. The number of partitions P can vary, therefore PWAH actually is a collection of schemes. Note that the number of partitions needs to be established beforehand, it is not possible to apply multiple PWAH compression schemes within a single bit vector.

In PWAH, every word consists of a header (using the P most significant bits) and P partitions. The header indicates the types of the different partitions: fill or literal. The partitions contain the actual representation of the uncompressed bits. Consider the following example word, based on a 64 bit word size and four partitions (PWAH-4):



The example 64 bit word consists of a header and 4 partitions:

0. Word header: four bits indicating the type of the four partitions. The first partition contains a fill (denoted by the 1-bit), the second partition contains literal bits (denoted by the 0-bit in the header), the third partition contains another fill and the fourth partition consists of a literal. Note that the header does not indicate the type of fill (1-fill or 0-fill).
1. First partition: the header indicates that this partition is a fill partition. The first (most significant) bit of the partition indicates the fill type: a 1-fill. The remaining 14 bits denote the length of the fill: 0b00000000001011 = 11 blocks. Since PWAH-4 uses a block size of 15 bits, 11 blocks represent a sequence of 165 1-bits.
2. Second partition: literal block (indicated by header 0-bit) with 15 literal bits.
3. Third partition: the header indicates a fill partition, the first bit of the partition determines the type: 0-fill. The length of the fill equals 0b00000000010010 = 18 blocks = 270 0-bits.
4. Fourth partition: another literal block containing 15 uncompressed bits.

The 64 bits shown in the example represent a total of $165 + 15 + 270 + 15 = 465$ bits. Note that the compressed bits are still aligned, hence the computational overhead of the compression (as described in [42]) is still reasonably low.

3.3.2 Number of partitions

Generally, it is assumed a word consists of 64 bits. To be able to use the 64 bits to their full extent, only a limited amount of partitions can be considered in order to use every single bit in a word. For example, consider a PWAH compression scheme using 13 partitions: 13 bits are allocated for storing the header and $64 - 13 = 51$ bits can be used to store partition information of all 13 partitions. Since partitions are required to have fixed and equal lengths, this scheme will contain 13 partitions of three bits, using a total of 39 bits. In this situation, 12 bits are left unused.

Let P denote the number of partitions, implying a header size of P bits. The remaining $64 - P$ bits should be divisible by P, yielding a positive integer result. Enumerating all possible values for P results in the following table:

P	**1**	**2**	3	**4**	5	6	7	**8**	9	10	⋯	15	**16**	⋯	**32**
$\frac{\mathbf{64-P}}{\mathbf{P}}$	**63**	**31**	$20\frac{1}{3}$	**15**	$11\frac{4}{5}$	$9\frac{2}{3}$	$7\frac{1}{7}$	**7**	$6\frac{1}{9}$	$5\frac{4}{10}$	⋯	$3\frac{4}{15}$	**3**	⋯	**1**

The divisions with integer results are typeset in bold. Clearly, all powers of 2 (up to and including $2^5 = 32$) qualify for P, which is not surprising:

$$\frac{64-P}{P} = \frac{2^6 - 2^k}{2^k} = \frac{2^6}{2^k} - \frac{2^k}{2^k} = \frac{2^6}{2^k} - 1 = 2^{6-k} - 1$$

It is evident that $2^{6-k} - 1$ yields a positive integer result for $k \leq 5$.

The numbers in the table shown above represent the number of bits allocated for a single partition. Although $2^{6-5} - 1 = 1$ bit (in the PWAH scheme with 32 partitions) is a perfectly valid number of bits, it is impossible to indicate both a fill type and a fill length using only a single bit. Therefore, the PWAH compression scheme with $P = 32$ partitions is considered to be invalid. The compression scheme with 16 partitions provides three bits per partition, of which two bits can be used to express a fill length. This results in a maximum fill length of three blocks (nine uncompressed input bits), which is considered to be impractical as well.

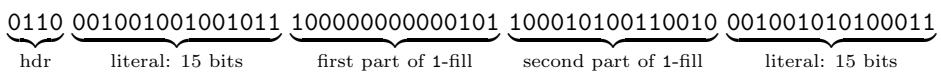
Note that the fill length in PWAH is expressed in a number of blocks, just like WAH. However, since the block size depends on the number of partitions, the number of single bits which can be represented by a fill depends on the number of partitions too. For example, the PWAH-4 fill `100000000001101` and the PWAH-8 fill `1001101` both represent 13 (`0b1101`) blocks of `1`-bits. Since PWAH-4 implies a block size of 15 bits, 13 blocks of `1`-bits actually represent 195 consecutive `1`-bits, whilst a `1`-fill of 13 blocks in PWAH-8 (with a block size of seven bits) actually contains 91 `1`-bits.

Figure 3.2 and Table 3.1 on page 27 provide an overview of the main properties of the different compression variations (including PWAH-16), based on a word size of 64 bits.

3.3.3 Extended fills

The major disadvantage of PWAH-8 and PWAH-4 over the regular WAH compression, is one of its advantages at the same time: the limited amount of bits which are allocated for expressing a fill length. Although in general one definitely does not need 30 bits to express a fill length, 14 bits (PWAH-4) or seven bits (PWAH-8) might turn out to be insufficient for compressing parts of certain bit vectors.

Of course it is possible to use two consecutive fill partitions to express a fill at most twice as large, but a more comprehensive solution would be to *concatenate* the bits from a number of partitions to express a fill length as if it were a *single* fill. We call this concept an *extended fill*: a fill of which the length is expressed by using multiple partitions. For example:



The second and third partition of the word shown in the example contain two consecutive 1-fills. Instead of interpreting the length of the 1-fills separately (0b00000000000101 + 0b00010100110010 = 5 + 1330 = 1335 blocks), the bits expressing the fill length are concatenated to denote:

$$\underbrace{\texttt{0b00000000000101}}_{\text{first part}}\underbrace{\texttt{01000110010}}_{\text{second part}} = 83{,}250 \text{ blocks}$$

The preceding 1-bit indicating the fill type was discarded. Note that there is no need to explicitly mark extended fills as such: each sequence of consecutive fills (of identical type) can be considered to denote an extended fill.

Using extended fills, the total number of bits which can be used to express a fill length increases from 14 to $4 \times 14 = 56$ bits (PWAH-4). This approach effectively eliminates the disadvantage of PWAH over WAH, whilst maintaining the advantage of providing a more fine grained compression scheme. The performance of the different PWAH compression schemes is evaluated in Chapter 4 on page 33.

3.3.4 Introducing the multi-OR operation

The paper by Wu *et al.* merely sketches a rough outline of a binary OR operator in WAH. However, in case a component C_k has more than two directly adjacent components $C_i, \ldots, C_j$, there are multiple orders in which the group OR of $C_i, \ldots, C_j$ can be computed by repeatedly applying the binary OR operator. Consider an example with three adjacent components:

$C_4 = (C_1 \vee C_2) \vee C_3$
$C_4 = (C_1 \vee C_3) \vee C_2$
$C_4 = (C_2 \vee C_3) \vee C_1$

The number of possible ways to process the input bit vectors grows extremely fast. Consider the following bit vectors containing PWAH-8 compressed reachability information:

$$C_1 = \underbrace{\texttt{00000000}}_{\text{header}}\underbrace{\texttt{0101010}}_{\text{literal}}\underbrace{\cdots\cdots\cdots\cdots}_{\text{6 more literals}}\underbrace{\texttt{0101010}}_{\text{literal}}$$

$$C_2 = \underbrace{\texttt{00000000}}_{\text{header}}\underbrace{\texttt{0101010}}_{\text{literal}}\underbrace{\cdots\cdots\cdots\cdots}_{\text{6 more literals}}\underbrace{\texttt{0101010}}_{\text{literal}}$$

$$C_3 = \underbrace{\texttt{1.....}}_{\text{header}}\underbrace{\texttt{1101111}}_{\text{large 1-fill}}$$

In the example depicted above, it would be a waste of time to compute the logical OR of C_1 and C_2 first, since C_3 contains a 1-fill of 15 blocks ($15 \times 7 = 105$ bits). Especially when the number of input components is large, it might be interesting to process multiple compressed bit vectors at the same time. After all, when encountering a large 1-fill in one of the input bit vectors, it is possible to skip large parts of the other bit vectors.

Another (practical) advantage of using MULTI-OR is the fact that it is no longer necessary to allocate memory to store intermediate results of the sequence of OR operations. However, this saving is only significant when merging a large number of large bit vectors.

Algorithm 4 on page 27 shows a small number of very high-level lines of pseudocode, implementing the multi-OR operation. Note that an actual implementation can become significantly more complex, the pseudocode merely shows the crux of the idea behind MULTI-OR. A few comments on the pseudocode:

l. 13 Within this *for* block, two tasks are carried out:

1. The bookkeeping variables belonging to the current input bit vector are updated to make sure the position within each input bit vector $B[i]$ (word index, partition index, partition offset, block index) matches the current position in the output bit vector r. For example, if a 1-fill from input bit vector $B[k]$ has been processed in the previous iteration of the

main *while* loop, it is necessary to skip the blocks in the other input bit vectors which were covered by the 1-fill from $B[k]$.

2. After having skipped over the input bit vector $B[i]$ to the correct position, it is possible to determine what kind of partition (fill or literal) is available next in $B[i]$. In order for the multi-OR to work as fast as possible, it is needed to take as large as possible steps: if a 1-fill is encountered, store its length if it is longer than the longest 1-fill encountered in this iteration. If a literal is encountered, compute the OR with the other literals from other input bitsets encountered earlier in this iteration. If a 0-fill is encountered, store its length if it is *shorter* than the shortest 0-fill encountered so far.

l. 22 If one of the input bit vectors provides a 1-fill, add that fill to the result and update the result block index *rbi* by adding the length of the fill.

l. 25 If a 1-fill is not available, but a literal was encountered in one or more of the input bit vectors: add the combined literal to the result and increase *rbi* by one (a literal is stored using a single block).

l. 28 If all input bit vectors point to 0-fills, the *shortest* 0-fill should be added to the result bit vector and *rbi* gets updated.

3.3.5 Runtime and memory usage

Although the experimental results in Chapter 4 show very promising results (regarding both compression ratio, construction time and lookup time), it is impossible to establish a meaningful asymptotic worst-case bound. The experiments show that the compression scheme yields a significant reduction of memory usage compared to other data structures for transitive closures, but it is quite trivial to construct a bit vector which is not suitable for compression at all. In such cases, PWAH (and WAH, which actually equals PWAH-2) will cause a small overhead which is linear in the length l of the original bit vector:

- PWAH-8: The header consists of eight bits, yielding an overhead of $\approx 14.29\%$;
- PWAH-4: The header consists of four bits, yielding an overhead of $\approx 6.67\%$;
- PWAH-2: The header consists of two bits, yielding an overhead of $\approx 3.23\%$ (equal to WAH);
- PWAH-1: The header consists of one bit, yielding an overhead of $\approx 1.59\%$.

Therefore, the memory usage of the compressed bit vector is $\mathcal{O}(l)$ (l denotes the number of bits in the original uncompressed bit vector).

A significant disadvantage of compressed bit vectors over regular bit vectors is the increased worst case lookup time. Regular bit vectors provide an $\mathcal{O}(1)$ look up, because bits are stored at a fixed offset in memory. However, after having compressed a bit vector, it is no longer possible to determine the position of a random bit within the bit vector within constant time. Given a bit index, a scan operation should be conducted considering every single partition. This results in a lookup time of $\mathcal{O}(w)$, w denoting the number of words in the compressed bit vector, growing linearly in the length l of the original uncompressed bit vector.

Setting and clearing bits is considered to be only possible for bits which have not yet been compressed by the RLE scheme (i.e. bits which still reside in a buffer). Under this assumption the operation can be performed in $\mathcal{O}(1)$ time. Altering a bit might involve processing (compressing) the buffer containing uncompressed bits, which can be done in constant time:

- Buffer consists of only 1-bits or 0-bits:
  1. Add a fill word to the bit vector or extend the length of the last fill;
  2. Set a 1-bit in the word header to indicate the partition contains a fill.
- Buffer consists of a non-contiguous (mixed) set of bits:
  1. Add a literal word to the bit vector;
  2. Set a 0-bit in the word header to indicate the partition contains a literal.

	Part. size (= block size)	**Maximum regular fill length**
PWAH-16	3 bits	$2^2 - 1 = 3$ blocks $= 9$ bits
PWAH-8	7 bits	$2^6 - 1 = 63$ blocks $= 441$ bits
PWAH-4	15 bits	$2^{14} - 1 = 16,383$ blocks $= 245,745$ bits
PWAH-2	31 bits	$2^{30} - 1 = 1,073,741,823$ blocks $= 33,285,996,513$ bits
PWAH-1	63 bits	$2^{62} - 1 = 4.61 \times 10^{18}$ blocks $= 2.91 \times 10^{20}$ bits

Table 3.1: Properties of PWAH compression schemes

Algorithm 4 multi-OR on PWAH compressed bit vectors. Section 3.3.4 on page 25 provides an explanation of the pseudocode

```
1: procedure MULTI-OR(B[0] ··· B[n − 1])
2:     iwi[0] ··· iwi[n − 1] ← 0          ▷ current word index of input bit vectors
3:     ipi[0] ··· ipi[n − 1] ← 0          ▷ current partition index of input bit vectors
4:     ipo[0] ··· ipo[n − 1] ← 0          ▷ current partition offset (blocks) of input bit vectors
5:     ibi[0] ··· ibi[n − 1] ← 0          ▷ current block index of input bit vectors
6:     r ← empty WAH bit vector           ▷ result
7:     rbi ← 0                            ▷ current block index in result bit vector
8:     while true do
9:         lof ← undefined                ▷ pointer to largest 1-fill
10:        szf ← undefined                ▷ pointer to smallest 0-fill
11:        lit ← 0                        ▷ OR-value of literals from input bit vectors
12:
13:        for all i ∈ {0 ··· n − 1} do
14:            align ibi[i], iwi[i], ipi[i] and ipo[i] to match rbi        ▷ See explanation
15:            if B[i] out of bounds then skip to next input bit vector
16:
17:            if B[i] contains largest 1-fill then lof ← B[i]
18:            if B[i] contains smallest 0-fill then szf ← B[i]
19:            if B[i] contains literal then lit ← lit ∨ B[i]
20:        end for
21:
22:        if lof is defined then
23:            add 1-fill to r
24:            rbi ← rbi + length of 1-fill
25:        else if lit is defined then
26:            add literal partition to r
27:            rbi ← rbi + 1
28:        else if szf is defined then
29:            add 0-fill to r
30:            rbi ← rbi + length of 0-fill
31:        else
32:            break                      ▷ no more input to process, abort while-loop
33:        end if
34:    end while
35:
36:    return r
37: end procedure
```

3.3.6 Adding indices

As was shown, the lookup time of a random bit within a PWAH bit vector takes linear time as opposed to the constant time it takes to look up a random bit in a regular bit vector. Introducing an index on a PWAH bit vector might improve the query time significantly.

Assuming that a division of l (number of uncompressed bits in the PWAH bit vector) can be performed in constant time, the process of looking up a random bit can be improved to $\mathcal{O}(k)$ time, where k denotes the chunk size of the index which does not depend on l. Therefore, the lookup time on an indexed compressed bit vector is considered to be constant. Introducing indices will increase memory usage of the compressed bit vector, but only linearly. Hence, the asymptotic memory usage does not change: $\mathcal{O}(l)$.

The concept of indices is illustrated using the following example representing 18,498,645 bits using two words of 64 bits (PWAH-4):

0100	001001001001011	100000001010001	100010100110010	001001010101100
hdr	literal: 15 bits	1-fill: 81 blocks	literal: 15 bits	literal: 15 bits
0110	001010100110010	000000001001011	001000100000101	001001010100011
hdr	literal: 15 bits	first part of 0-fill	second part of 0-fill	literal: 15 bits

Using the following index (with a chunk size of 1024 bits) it is possible to optimise the lookup of random bits in the compressed bit vector, by being able to instantaneously jump to a partition close to the bit:

index position	1	2	3	4	$\cdots$	18,064	18,065
index bit	1,024	2,048	3,072	4,096	$\cdots$	18,497,536	18,498,560
global block	68	136	204	273	$\cdots$	1,233,169	1,233,237
word	0	1	1	1	$\cdots$	1	1
partition	1	2	2	2	$\cdots$	2	2
block within part.	67	51	119	188	$\cdots$	1,233,084	1,233,152

Every index entry points out where a bit with a specific index (multiples of k) resides within the compressed bit vector. For example, bit 1024 resides within word 0, partition 1, block 67 (note that the 1-fill partition contains 81 blocks). Now that the bit vector is indexed, a query regarding some bit j would no longer require a full scan over all partitions. It is possible to look up the closest smaller indexed bit i in the index by looking at index position $\lfloor \frac{j}{k} \rfloor$ and commence seeking from the partition in which i resides, until j is encountered. This would require processing at most $\frac{k}{b}$ (b denoting the block size, 15 in PWAH-4) partitions.

Although introducing indices might reduce the asymptotic time complexity, it is hard to predict whether it will actually have a noticeable positive effect on the performance of lookups. It might very well be that indices introduce an overhead which cancels out a possible positive effect. This will be subject of further evaluation in Chapter 4.

3.3.7 Theoretical analysis of memory usage

Introduction

Although the experiments show interesting results, it is very hard to provide a decent theoretical foundation for PWAH with a meaningful worst-case analysis. The main difficulty lies within the fact that its performance largely depends on the topology of the input graph. However, people have been using interval lists for quite some time [24] and PWAH shows a very clear resemblance to interval lists when it comes to compression performance. Both approaches highly depend on the graph topology and the order in which vertices are processed, both are excellent in compressing large sequences of consecutive 1-bits or 0-bits. Therefore, a short analytical comparison of compression performance is presented in this section.

Worst case PWAH

An interval list can contain intervals starting at arbitrary positions and of arbitrary length: $I = \{(i_0, j_0), (i_1, j_1), \ldots, (i_{n-2}, j_{n-2}), (i_{n-1}, j_{n-1})\}$. It is clear that a single interval resembles a fill partition in PWAH, but only under certain circumstances it is possible to represent a single interval by a single fill partition. Consider the following example interval list I consisting of a two intervals and the PWAH-4 bit vector B representing the same information:

$I = \{(5, 49), (94, 101)\}$

$B =$ `0101` `000001111111111` `100000000000010` `111110000000000` `000000000000010`

hdr	literal: 15 bits (bits 0-14)	1-fill: 2 blocks (bits 15-44)	literal: 15 bits (bits 45-59)	0-fill: 2 blocks (bits 60-89)

`0000` `000011111111000` `000000000000000` `000000000000000` `000000000000000`

hdr	literal: 15 bits (bits 90-104)	unused partition	unused partition	unused partition

Although the interval list consists of two intervals, it is most certainly not possible to represent these two intervals by two fill partitions. Firstly, the space in between the intervals needs to be represented by a 0-fill and secondly, a fill can not start at any arbitrary position within the bit vector, but can only start at block boundaries. Worst case, a single interval needs to be represented by a preceding 0-fill, a literal, a 1-fill and a literal. An example with three intervals can be illustrated like this:

$I = \{$ (␣␣…␣␣) (i_0, j_0) (␣␣…␣␣) (i_1, j_1) (␣␣…␣␣) (i_2, j_2) $\}$

0-fill	lit.	1-fill	lit.	0-fill	lit.	1-fill	lit.	0-fill	lit.	1-fill	lit.
1st interval				2nd interval				3rd interval			

In this worst case scenario, a single interval will require four PWAH partitions – assuming that the length of both the 1-fill and the 0-fill do not require one or more extended fills (and span more than one partition each). Table 3.2 on page 31 contains an overview of the different types of PWAH and the amount of bits required to represent a single interval as described above.

A straightforward implementation of an interval list would use two 32-bit integers to store a single interval. Looking at Table 3.2, it is immediately clear that PWAH-4, PWAH-2 and PWAH-1 will always require as least as much memory to store an interval as an interval list would use in this worst case scenario. If the two fills can be stored using a single partition each, PWAH-8 would use half as much memory as an interval list. Actually, PWAH-8 allows using up to six partitions for representing the fill length of the 1-fill and 0-fill together, without using more memory than equivalent interval lists would use.

Theorem 1. *The number of partitions p required to store a fill consisting of n bits using a PWAH compression scheme with block size b equals:*

$$p = \left\lceil \frac{\lceil log_2(\frac{n}{b} + 1) \rceil}{b - 1} \right\rceil$$

(assuming n is a multiple of b)

Proof. The fill length is expressed in multiples of b, rather than in single bits. Therefore, the number which needs to be encoded as the fill length equals $\frac{n}{b}$. The number of bits required to express that length equals $\lceil log_2(\frac{n}{b} + 1) \rceil$. In a partition consisting of b bits, $b - 1$ bits are available for storing a fill length. □

Assuming an interval list uses 64 bits to store a single interval, it is possible to create a visual representation of the interaction between PWAH-8 and interval lists memory usage. The two plots depicted in Figure 3.3 on the next page illustrate this interaction:

Figure 3.3a – The number of bits within the 0-fill preceding the interval and the size of the 1-fill representing the interval itself are depicted on the x and y axes, the z axis shows the number of partitions used to store an interval. The green surface indicates the total amount of partitions used by the PWAH-8 scheme to store an interval, which includes storing the preceding 0-fill and both the two preceding and succeeding literal partitions. It is clear that the z value of the green surface highly depends on the lengths of the 0-fill and 1-fill. The orange plane depicts the amount of memory to store a single interval in an interval lists: a constant 64 bits (or, as depicted on the z-axis: 8 partitions). The equation used to generate the green surface in Figure 3.3a is:

$$f(x,y) = \frac{log_2(\frac{x}{7}+1)}{6} + \frac{log_2(\frac{y}{7}+1)}{6}$$

Based on Theorem 1 on the preceding page.

Figure 3.3b – This figure depicts the dependency of the size of the 0-fill and the 1-fill, given the fact that the number of partitions per interval is limited by eight in order to operate more efficiently than an interval list. The graph is symmetrical (like Figure 3.3a), hence it does not matter to which of the fills the x and y axes belong: the more bits one of the fills needs to store, the less the other fill can contain without using more than eight partitions.

As can be concluded from both figures, only a combination of an extremely large preceding 0-fill and an extremely large 1-fill will cause PWAH-8 to use more than eight partitions to represent a single interval. Therefore, it is expected that PWAH-8 will perform better than interval lists in terms of memory usage in virtually all cases. Whether this expectation is actually met or not when using real-world graphs will be investigated in Chapter 4.

Worst case interval lists

The worst case situation for the PWAH scheme (compared to interval lists) has been shown in the previous section. Obviously, it is also interesting to have a brief look at the opposite situation: the worst case scenario for interval lists with respect to compressed bit vectors.

Although the PWAH scheme has great difficulty handling sequences containing 'scattered' bits, it will still use only one bit for every plain bit. Interval lists on the other hand, will require 64 bits to represent a single 1-bit from a bit vector. For example, consider the following uncompressed scattered bit vector B_u consisting of 16 bits, the equivalent PWAH-8 compressed bit vector B_c and the equivalent interval list:

B_u= 01010101 01010101 $\cdots$
B_c= $\underbrace{000\ldots..}_{\text{header}}$ $\underbrace{0101010}_{\text{literal}}$ $\underbrace{1010101}_{\text{literal}}$ $\underbrace{0100000}_{\text{literal}}$ $\cdots$

$I = \{(1,1),(3,3),(5,5),(7,7),(9,9),(11,11),(13,13),(15,15)\}$

In this situation, an interval lists would require $8 \times 2 \times 32$ bits = 512 bits to represent the same information as the bit vectors contain. However, as explained in Section 2.2.5, it is very unlikely that these worst case intervals would emerge from a graph when performing a DFS.

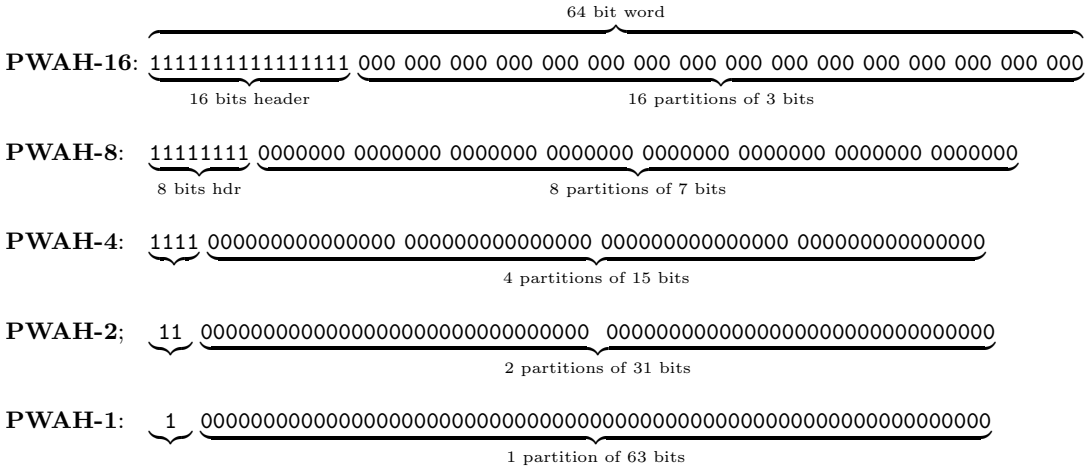
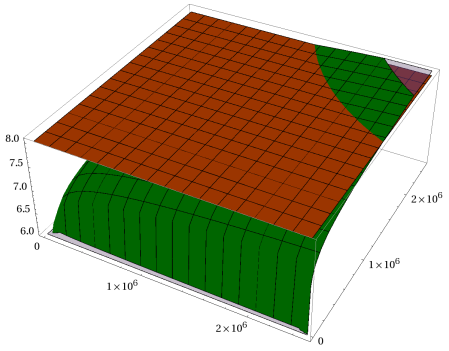


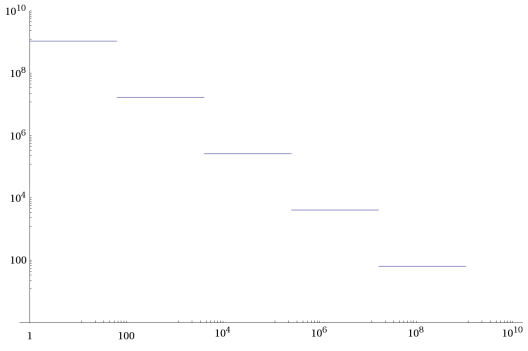
Figure 3.2: All 5 PWAH schemes: 1, 2, 4, 8 and 16 partitions

	BS	# part. 0-fill and 1-fill	Total # part.	Bits
PWAH-8	7	$1+1=2$	4	$4\times 7+4=32$
	7	3	5	$5\times 7+5=40$
	7	4	6	$6\times 7+6=48$
	7	5	7	$7\times 7+7=56$
	7	6	8	$8\times 7+8=64$
PWAH-4	15	$1+1=2$	4	$4\times 15+4=64$
	15	3	5	$5\times 15+5=80$
	15	4	6	$6\times 15+6=96$
PWAH-2	31	$1+1=2$	4	$4\times 31+4=128$
	31	2	5	$5\times 31+5=160$
PWAH-1	63	$1+1=2$	4	$4\times 63+4=256$

Table 3.2: Overview of different PWAH schemes and the number of bits required to store an interval



(a) A 3D plot illustrating how interval lists and PWAH-8 interact



(b) A plot illustrating the dependency of the 1-fill and 0-fill in PWAH-8

Figure 3.3: Plots illustrating the relation between interval lists and PWAH-8

Chapter 4

Experimental evaluation

4.1 Set up of experiments

4.1.1 Introduction

Although run time complexity has been analysed in Chapters 2 and 3, an experimental analysis using varying inputs might provide a better understanding of the actual behaviour of the algorithms and data structures. In order to obtain such insight, both real-world and randomly generated graphs were used to analyse the performance of the data structures.

In particular, the behaviour of the newly introduced PWAH data structures has been examined very carefully. The structures were compared to both interval lists, Path-Tree (as described in Section 2.3.2, [19]) and 3-Hop (Section 2.3.3, [18]).

4.1.2 Experiments of interest

This focus of this experimental evaluation is twofold:

1. determine which influence the input graphs have on the performance of the algorithms and data structures;
2. determine the performance of PWAH compared to existing other approaches.

It is expected that the influence of the input graphs highly depends on the main characteristics of the graph: the graph topology, the number of vertices and the number of edges. Since all approaches highly depend on the structure of the condensation graph, the number of strongly connected components and the number of edges between the components is expected to be of major importance as well.

When evaluating 'performance', three aspects are considered to be of interest:

1. processing time: how long does it take to build a transitive closure data structure?
2. query time: how much time does it take to answer a reachability query using the data structure?
3. memory usage: how much memory does the finished data structure consume?

4.1.3 Random graphs

Introduction

To be able to determine the influence the input graphs have on performance, a large number of randomly generated graphs have been used. The main properties (number of vertices and edges) of the graphs can be determined beforehand, which provides the opportunity to have a very close look at possible correlation between these graph properties and performance. Unfortunately, it is very hard to generate graphs with a specific number of strongly connected components and inter-component edges without losing the property of proper randomness. Very advanced techniques

exist (as described in e.g. [22, 23]) to generate graphs which are clustered to some extent, but these methods lie far beyond the scope of this thesis.

Generating random graphs

The process of generating a random graph starts with determining the desired number of vertices and inserting these vertices into the graph data structure. Adding edges is a more elusive process and can be done in a number of ways:

1. Iterate over the set of all possible edges $\mathbb{E} = V \times V$, select edges with a probability p and insert the edges into edge set. This approach results in $|E|$ scaling with $|V|$.
2. Determine the desired out-degree of a vertex and randomly pick neighbour vertices. Using this approach, $|E|$ does not scale with $|V|$. Note that it is possible to establish a probability distribution which determines the out degree of a vertex. Such a distribution can have any type, e.g. a uniform or normal distribution. Using this approach to selecting edges, a vertex can have any arbitrary in-degree.
3. Determine both an in-degree and out-degree (by using any kind of probability distribution) for each vertex and add incoming and outgoing *edge stubs* to each vertex. Randomly pick in-stubs and out-stubs and connect these to form a new edge. At some point, there will be no in-stubs or out-stubs left and the process terminates. Although this method does not guarantee an exact in-degree or out-degree for vertices, it does provide a way to pose a limit to the degrees of vertices, which will result a more uniform graph in terms of degrees.
4. Determine the number of vertices and edges, randomly pick pairs of vertices from V and connect the pair using an edge, until the desired number of edges has been added.

As expressed before, showing a possible correlation between performance on one side and the number of vertices and edges on the other is one of the two pillars of this experimental analysis. Therefore, method 4 described above was used to generate a large data set of random graphs, since it provides us with a way to influence the number of edges and vertices in a direct way.

Characteristics of generated graphs

The graph generation methods listed above are not designed to produce graphs which resemble or model any specific type of real-world network. The number of strongly connected components is obviously strongly correlated to the number of vertices and edges: a graph with a relatively large number of edges will inevitably have a small number of components.

The plot depicted in Figure 4.1 on the next page shows the number of random graphs (Z axis) obtained, with the number of strongly connected components and edges between the components on the X and Y axes. The graph has been cropped in the direction of the Z axis in order to be able to show the most relevant part. The number of graphs with $\leq 10,000$ strongly connected components and $\leq 10,000$ edges between the components is close to 4,000.

4.1.4 Non-artificial graphs

To illustrate the variety of graph types for which bit vector compression provides a significant advantage over existing data structures, a large number of graphs from different sources and fields of research has been used as input for the experiments which were conducted. The graphs can be divided into multiple groups:

- Source code analysis graphs provided by Semmle Ltd. This set consists of a large number of graphs used for different kinds of source code analysis of multiple open source projects:

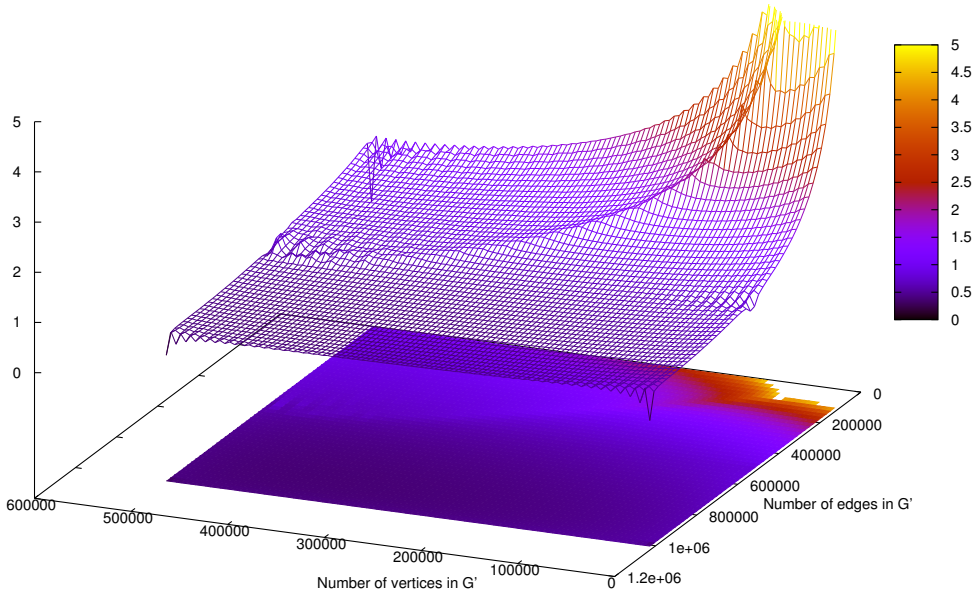


Figure 4.1: Correlation between the number of vertices and edges in the condensation graph G' on the X and Y axes and the number of generated graphs on the Z axis.

- ADempiere [2] (3.5.1a), an open source ERP application written in Java.
  - *calls* Describes direct calls between methods;
  - *child* Represents the parent-child relation in the abstract syntax tree;
  - *depends* Represents the dependency relation between types;
  - *polycalls* Describes direct calls between methods, taking the virtual dispatch into account;
  - *subtype* Contains the subtype relationship of classes and interfaces;
  - *successors* Models the control-flow successor relation between statements.
- ImageMagick [8] (6.5.7-8), a collection of applications to display and edit raster images.
  - *callgraph* Describes direct calls between methods, using points-to analysis to resolve virtual calls;
  - *depends* Describes the dependencies between source files;
  - *successors* Models the control-flow successor relation between expressions.
- Samba [36] (3.2.23), an open source server for the SMB/CIFS network protocols.
  - *setflow* Contains inclusion sets used to perform a points-to analysis.

- Wikipedia structural graphs provided by Semmle Limited (based on the simple English Wikipedia as of the 1st of July, 2009).
  - *categorylinks* Links between category pages;
  - *categorypagelinks* Links from category pages to any other page (category or regular);
  - *pagelinks* Links between pages in Wikipedia.

- Graphs used by Jin *et al.* in [19]:

AgroCyc, Anthra, Ecoo, HpyCyc, Human, Mtbra and VchoCyc	Graphs originating from the EcoCyc project [10], a scientific database for the bacterium *Escherichia Coli* K-12 MG1655;
Xmark and Nasa	XML documents (not credited in [19]);
Reactome, aMaze and KEGG	Metabolic networks provided by Trißl *et al.* [37].

- Graphs used by Jin *et al.* in [19]:

ArXiv	Subset of data set of citations within research papers, as found on the ArXiv website [38];
CiteSeer	Citations from database of CiteSeer project website [15];
pubmed	Citations in publications from the medical research field, as found on the PubMed website [39];
GO	Data set of genetic terms from the Gene Ontology project [35];
YAGO	Relationships within the semantic knowledge database from the YAGO project [33].

- Road graph:

benelux	A road network graph of Belgium, The Netherlands and Luxembourg.

Characteristics of these graphs are listed in Table 4.1 on the facing page.

4.1.5 Environment and tools

Machine

All experiments were carried out on a 64 bit Intel® Core™2 Quad CPU type Q6700 with four cores at 2.66GHz. A total of four gigabytes of memory was installed.

Operating system

The experiment machine was running Ubuntu 10.04 LTS with Linux kernel 2.6.32. The operating system was stripped of any graphical interface and only core services were still running in order to be able to obtain reliable timings.

Tools

The GNU C++ compiler (version 4.4.3) was used to compile C++ code. Version 4.4.0 of the gnuplot tool was used to visualise the obtained data in the plots presented in this experimental evaluation.

4.2 Influence of input graphs on PWAH performance

4.2.1 Construction time: a short preface

The Nuutila/PWAH approach to transitive closure computation has two main computational tasks:

1. Graph traversal and detection of strongly connected components. The number of edges and vertices will most likely have a significant influence on the time spent on this task;
2. Merging reachability information of strongly connected components. The number of strongly connected components and the number of edges between the components will most likely have a significant influence on the time spent on this task.

Due to the fact both tasks are performed in one graph traversal, it is very hard to analyse the time spent on either of the two tasks in isolation.

	$\|V\|$	$\|E\|$	$\|V_C\| = \|C\|$	$\|E_C\|$	$\|E^+\|$	$\|E^*\|$
ADempiere [2]						
Calls	52,290	198,041	52,225	8,726,099	8,972,548	9,024,678
Child	181,091	180,373	181,091	568,468	568,468	749,559
Depends	6,332	79,804	4,679	2,205,417	7,756,079	7,758,212
Polycalls	53,206	595,198	44,839	159,722,084	438,508,149	438,552,734
Subtype	10,391	32,588	10,391	92,252	92,252	102,643
Successors	423,358	448,909	384,094	4,014,609	7,323,367	7,703,798
ImageMagick [8]						
Callgraph	3,685	22,282	3,340	867,109	1,787,728	1,791,047
Depends	524	11,408	278	13,441	128,143	128,240
Successors	1,095,062	1,145,304	542,235	120,800,042	2,309,714,078	2,310,250,306
Samba [36]						
Setflow	568,656	884,839	510,720	3,134,420,405	10,968,067,063	10,968,568,966
Wikipedia						
Categorylinks	11,808	18,630	11,798	230,116	234,507	246,288
Cat. pagelinks	75,946	181,084	75,936	2,209,952	2,262,118	2,338,037
Pagelinks	137,830	2,949,220	47,242	104,513,953	12,479,685,213	12,479,732,188
Jin *et al.* [19] 2008						
AgroCyc	13,969	17,694	12,684	170,591	2,731,596	2,744,279
Amaze	11,877	28,700	3,710	2,371,476	93,685,094	93,688,747
Anthra	13,736	17,307	12,499	148,559	2,440,124	2,452,621
Ecoo	13,800	17,308	12,620	173,254	2,398,250	2,410,868
HpyCyc	5,565	8,474	4,771	77,131	1,113,356	1,118,124
Human	40,051	43,879	38,811	348,112	2,804,552	2,843,362
Kegg	14,271	35,170	3,617	2,637,440	147,922,066	147,925,460
Mtbrv	10,697	13,922	9,602	139,275	2,038,237	2,047,836
Nasa	5,704	7,939	5,605	167,243	186,900	191,250
Reactome	3,678	14,447	901	32,607	6,666,940	6,667,788
VchoCyc	10,694	14,207	9,491	136,674	2,343,636	2,353,125
Xmark	6,483	7,954	6,080	536,389	2,307,574	2,313,653
Jin *et al.* [18] 2009						
ArXiv	6,000	66,707	6,000	5,566,205	5,566,205	5,572,205
CiteSeer	10,720	44,258	10,720	421,995	421,995	421,995
GO	6,793	13,361	6,793	104,178	104,178	104,178
PubMed	9,000	40,028	9,000	523,037	523,037	523,037
YAGO	6,642	42,392	6,642	66,439	66,439	66,439
Road network						
Benelux	1,598,250	3,756,335	406	39,295	2,553,615,187,901	2,553,615,188,285

Column definitions:

$|V|$ = Number of vertices;
$|E|$ = Number of edges;
$|V_C| = |C|$ = Number of strongly connected components (vertices in the condensed graph);
$|E_C|$ = Number of edges between strongly connected components (edges in the condensed graph);
$|E^+|$ = Number of edges in the *regular* transitive closure;
$|E^*|$ = Number of edges in the *reflexive* transitive closure.

Table 4.1: Characteristics of non-artificial graphs used for experiments

4.2.2 Construction time: vertices and edges

Figure 4.2 contains three plots depicting the construction time of a transitive closure data structure with respect to the number of vertices and edges of a directed graph G. Several observations can be made and questions can be raised based on these figures.

First, the limited range of the plot lines sticks out. During the graph generation process, graphs containing less than ten strongly connected components (or 50, for $|V| > 100,000$) are considered to be not of any interest and are therefore not stored. The number of strongly connected components highly depends on the number of edges: an increasing number of edges yields a strongly decreasing number of strongly connected components and therefore, the plot lines abruptly end.

The next remarkable characteristic is the maximum located between $|E| = 20,000$ and $|E| = 700,000$, depending on the number of vertices. This phenomenon is caused by the fact that the total amount of time spent on merging reachability information highly depends on the number of strongly connected components in the input graph. Graphs with a small number of edges (relative to the number of vertices), are expected to contain a number of strongly connected components which roughly equals the number of vertices, i.e. most components consist of only one vertex. Since the operation of merging reachability information is (1) carried out exactly once for every component and (2) depends on the number of reachable strongly connected components, the processing time will initially increase drastically when adding edges.

After reaching a critical edge density, randomly adding more edges will result in creating strongly connected components consisting of more than just one vertex. As the number of edges increases, the number of components drops drastically and the number of required merge operations will decrease. This effect can clearly be seen in the plot figures, as the plot lines drop to a local minimum somewhere between $|E| = 60,000$ and $|E| = 1,500,000$ (again, depending on the number of vertices).

At the local minimum, the number of strongly connected components is still decreasing, but only very slowly. Henceforth, the time gained by having to perform less merge operations does no longer dominate the costs of (1) graph traversal and (2) storing more and more reachability information induced by the increasing amount of edges. After a while, the number of components falls below the threshold and the plot lines end.

The observation regarding the maxima in the plots illustrates the influence of both of the main computational tasks on the total processing time. By carefully looking at the third plot (Figure 4.2c) and comparing it to the first one (Figure 4.2a), the explanation of this observation can be validated: although different compression schemes were used to generate the plots and therefore the maxima are located at significantly different locations, all plot lines end at roughly the same values of construction time (for example, 700 milliseconds for $|V| = 500,000$ and $|E| = 6,000,000$).

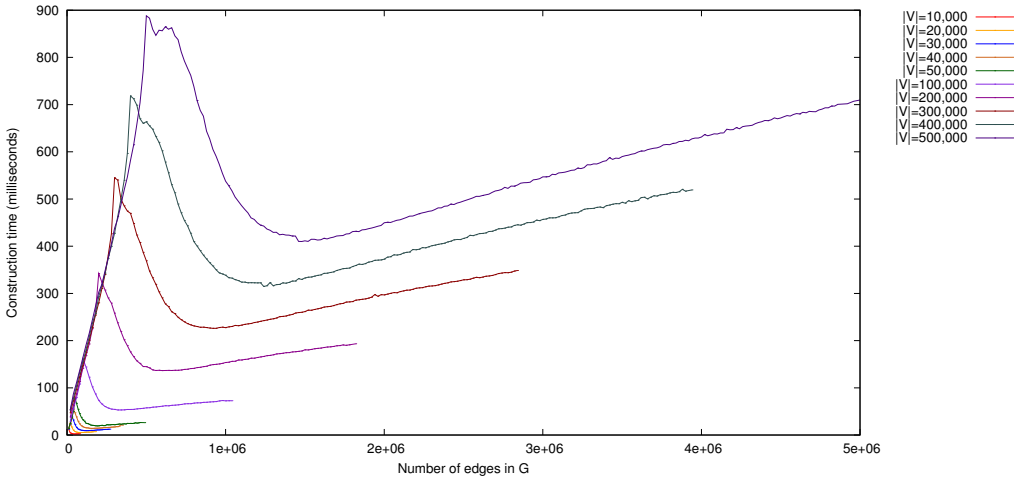
This confirms that the time spent on merging reachability information (which is obviously significantly influenced by the choice of compression scheme or data structure) highly depends on the number of strongly connected components. Once the number of components has shrunk sufficiently, the time spent on merging reachability information no longer dominates the time used by the DFS process to traverse the graph.

Note that the last observation at the same time explains the apparently linear correlation between the number of edges and the processing time (for larger values of $|E|$): at some point, the linear DFS procedure dominates the time spent on computing reachability information.

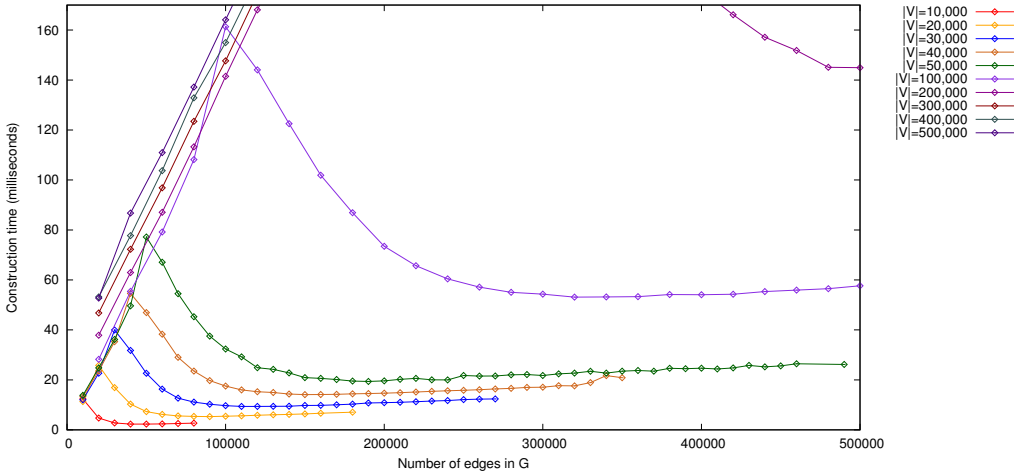
4.2.3 Construction time: vertices and edges in condensation graph

The plots in Figure 4.2 suggests a correlation between the number of components and the construction time of the transitive closure data structure. Figure 4.3 on page 40 is based on three similar generated graphs and depicts two correlations within each of the three plots:

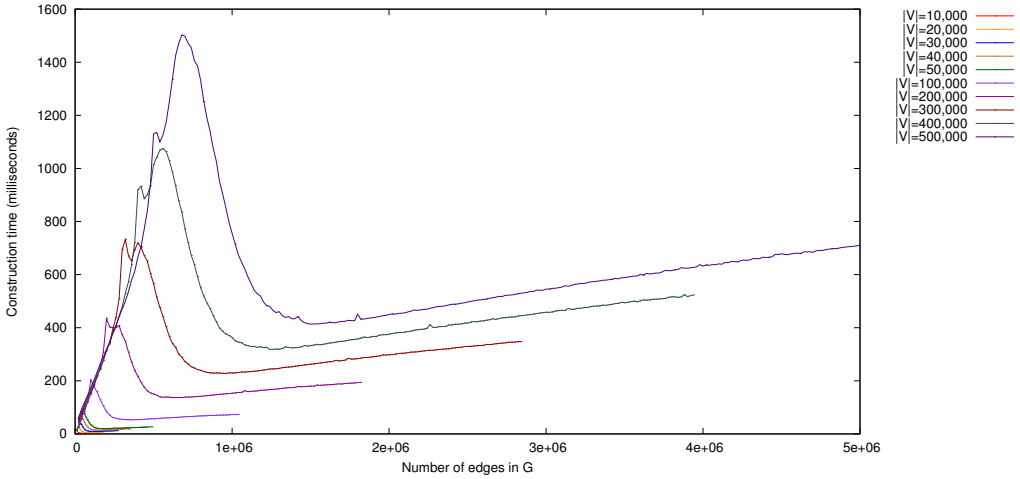
1. the construction time with respect to the number of strongly connected components in the input graph, and



(a) Construction time of TC data structure using PWAH-2 w.r.t. number of vertices and edges

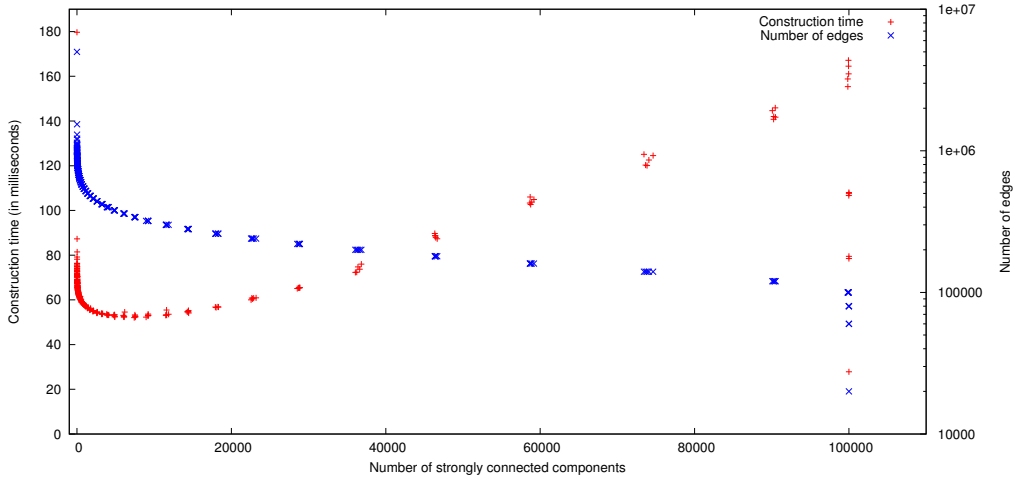


(b) Construction time of TC data structure using PWAH-2 w.r.t. number of vertices and edges, detail of plot shown above

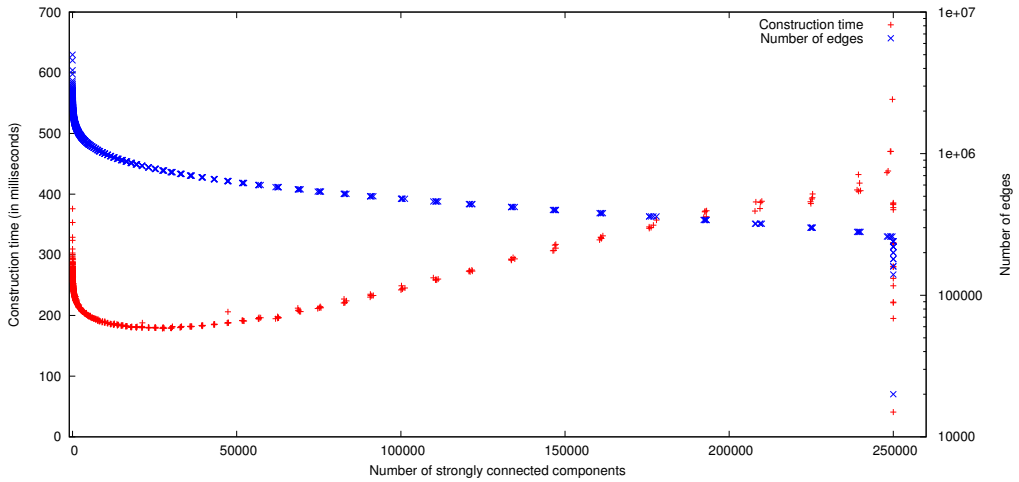


(c) Construction time of TC data structure using PWAH-8 w.r.t. number of vertices and edges

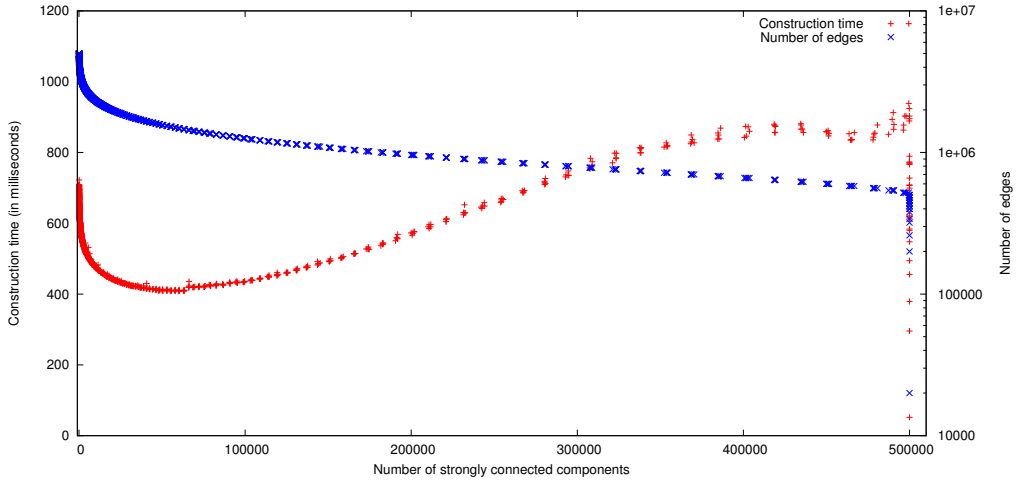
Figure 4.2: Plots of construction time (Y axes) using PWAH-2 and PWAH-8 with respect to the number of edges in a graph (X axes).



(a) $|V| = 100,000$



(b) $|V| = 250,000$



(c) $|V| = 500,000$

Figure 4.3: Plots of construction time (using PWAH-2, depicted using red points: $+$) and the number of edges (blue points: $\times$), based on randomly generated graphs with fixed vertex counts and up to 5,000,000 edges.

2. the number of edges required to yield a specific number of strongly connected components.

The construction time of the transitive closure data structure indeed turns out to be strongly correlated to the number of strongly connected components (denoted by $|C|$) in the input graph. For very small and very large values of $|C|$, the construction time appears to be very unpredictable: between 180 and 50 milliseconds for a small number of components, between 30 and 170 milliseconds for graphs containing a large number of components.

This phenomenon can again very easily be explained by the relation between the number of edges and vertices and the number of strongly connected components. Note that the number of vertices is fixed within each graph: 100,000 vertices in Figure 4.3a, 250,000 in Figure 4.3b and 500,000 vertices in Figure 4.3c. For randomly generated graphs containing $|V| = 100,000$ vertices, the number of strongly connected components roughly equals 1 when $|E| \geq 1,300,000$. On the other end, the number of components equals 100,000 when $|E| \leq 80,000$. Since the number of edges was increased linearly when generating the graphs, a large number of data points lie on the far left and far right side of the plot.

As indicated, the number of edges in these graphs (depicted on the logarithmically scaled secondary Y axes by using the red points: +) determines the number of resulting strongly connected components. The two data series are displayed on the two Y axes within the same plots to show the remarkable resemblance between them for smaller values of $|C|$.

4.2.4 Query time

To measure the effect of the size of the input graph (in terms of numbers of edges and vertices) on the query time of the PWAH data structure, one million source and destination vertices were randomly selected. The total time to answer one million reachability queries on a graph with 500,000 vertices was measured and is depicted in Figure 4.4 on the following page.

As has been explained earlier: a large amount of data points is located at both ends of the X axis. As a very large and diverse range of graphs is responsible for these data points, it is impossible to draw conclusions based on these values. The general shape of the plot indicates a linear correlation between the number of components of a graph and the time it takes to answer 1,000,000 random queries on the associated transitive closure data structures. This correlation was to be expected: the length of the reachability bit vectors will grow roughly linearly with the number of strongly connected components. Since a single reachability query requires a linear scan over a PWAH compressed bit vector, the time to answer such a reachability query is expected to grow linearly with the length of the bit vector.

4.2.5 Memory usage

The plots in Figure 4.5 on page 43 show the number of bits used by PWAH-2 with respect to the number of strongly connected components in the input graph. Note that the number of vertices in the input graph is fixed to 500,000, the number of edges varies from 10,000 to 5,000,000. Once again, the data points to the far left and right are based on a very large number of graphs with varying properties. Hence, it is impossible to draw conclusions based on these points.

The general shape of the first plot hints towards a very subtle quadratic (actually close to linear) correlation between the number of bits required to store the transitive closure representation and the number of components. The second plot contains the same data points, but has been drastically rescaled to fit the number of bits an equivalent reachability matrix would use to store the same reachability information.

4.3 Comparing different PWAH schemes

The performance of the different PWAH compression schemes is compared in Figure 4.6 on page 44 by plotting the construction time, query time and memory usage of the three schemes. The relative performance of the three different compression schemes does not show any unexpected results:

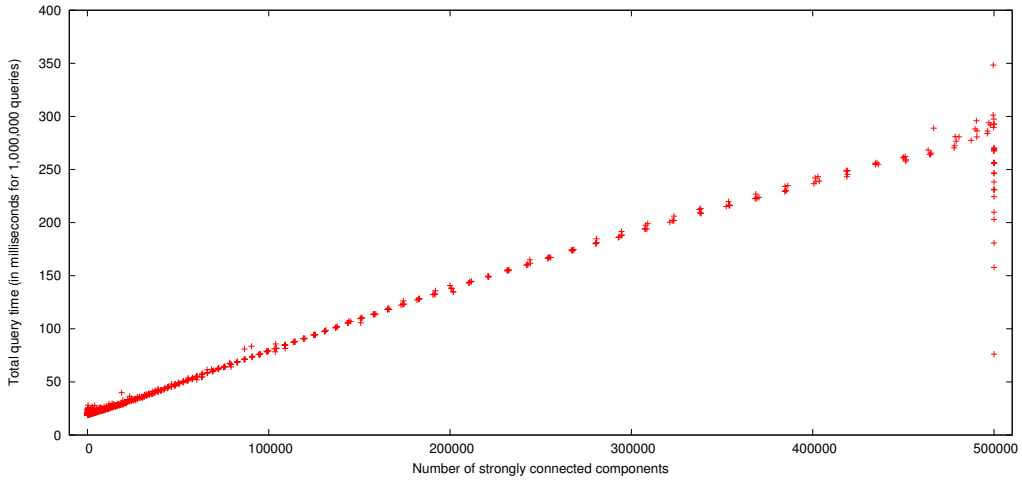


Figure 4.4: Total time required to answer 1,000,000 queries with respect to the number of strongly connected components in random graphs with 500,000 vertices and up to 5,000,000 edges using PWAH-2

PWAH-8 implements a more fine grained compression and is therefore uses less memory but is slightly slower than PWAH-2 and PWAH-4.

Figure 4.6a does show a remarkable local maximum at $|C| = 350,000$, which can most clearly be seen in the PWAH-8 data points (depicted in orange: ✳). For a number of strongly connected components larger than 350,000, the total construction time actually starts decreasing. This effect can be explained by the way the random graphs were generated: a large number of strongly connected components in a graph with 500,000 vertices indicates the graph contains a relatively small number of edges. Less edges imply less reachability, resulting in PWAH bit vectors consisting of mostly 0-fills. As explained in Section 3.2, PWAH bit vectors consisting of large fills can be processed in a very efficient way. Therefore, although the number of strongly connected components is increasing, the total time spent on merging reachability information is actually decreasing.

4.4 Indexing PWAH

As explained in Section 3.3, the main disadvantage of PWAH over a regular (uncompressed) bit vectors is the fact that performing a lookup operation requires linear time. In order to reduce the asymptotic time required for such an operation, an index on the PWAH bit vector was introduced. Figure 4.7 on page 46 shows three plots illustrating the effect of applying such indices to the performance of PWAH-8 on a graph with 500,000 vertices and up to 5,000,000 edges.

The figure depicts the results of multiple *index chunk sizes* (as introduced in Section 3.3.6), denoted by i. For example, $i = 1024$ means the location of every 1024^{th} bit in the bit vector is explicitly stored in the index. A larger chunk size results in a less locations being stored in the index and therefore a smaller index size.

The results of the experiments with indices are rather surprising. One would expect to see two trends:

1. indexing the bit vector yields a constant query time;
2. a more fine grained index will yield a better query performance.

Looking at the results of the experiments, both expectations are defied. It is very hard – if not impossible – to determine exactly why indexing the PWAH vectors turns out to work counterproductively, but the most likely cause is to be found within caching behaviour. Whilst using computer memory, the operating system will precache the memory which is most likely to be read in the very

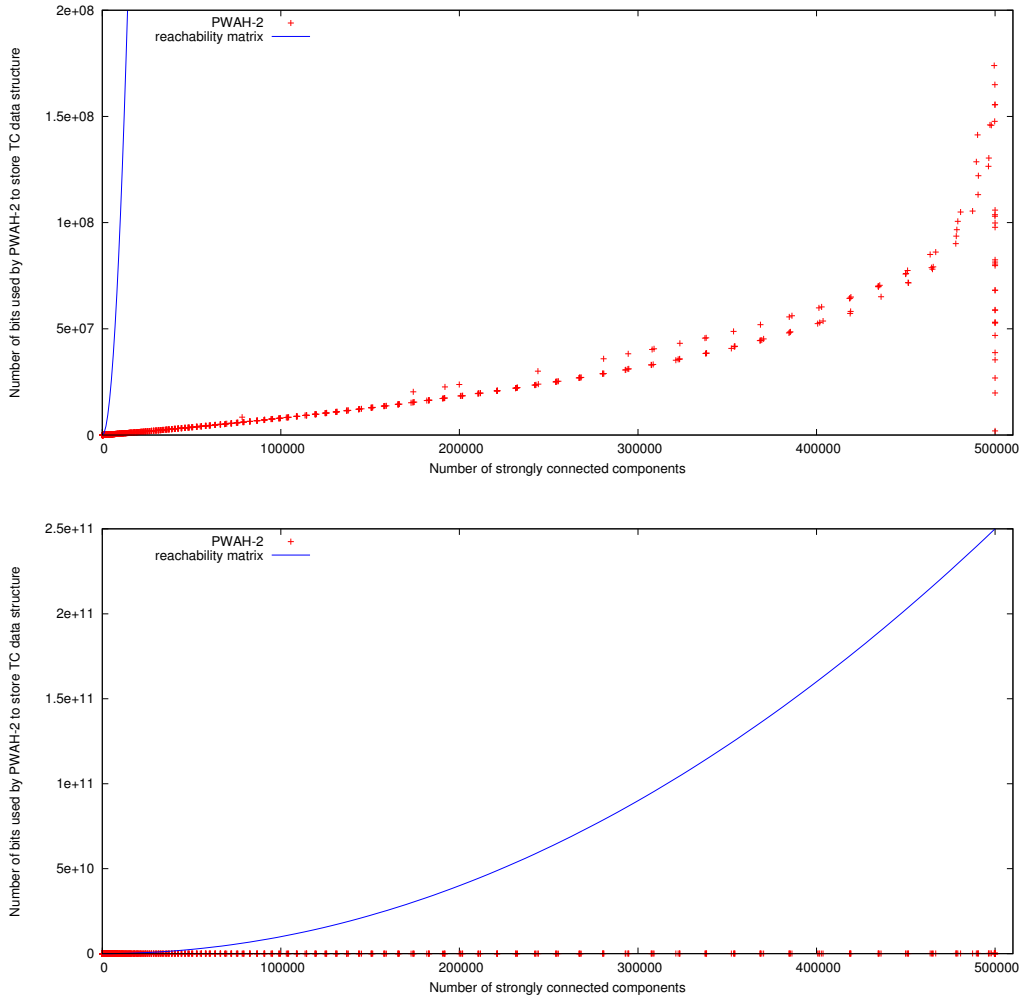
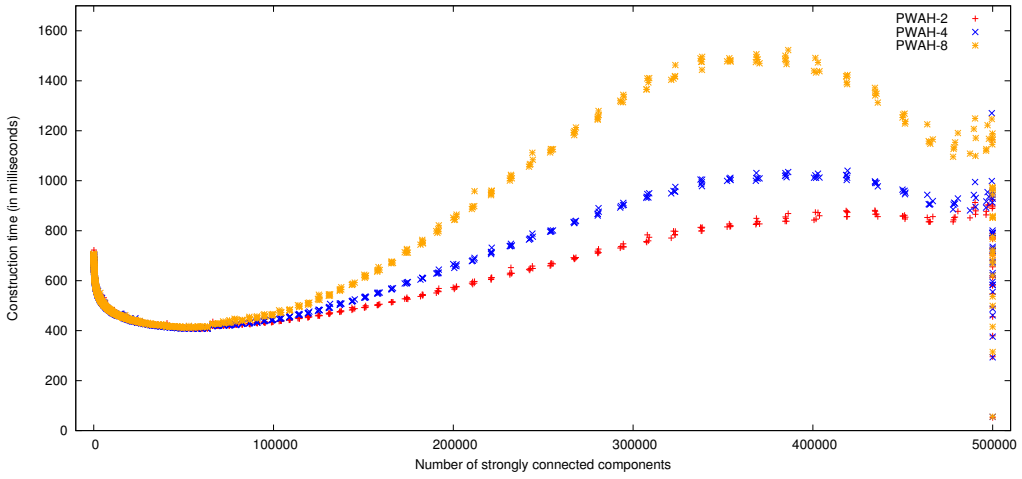
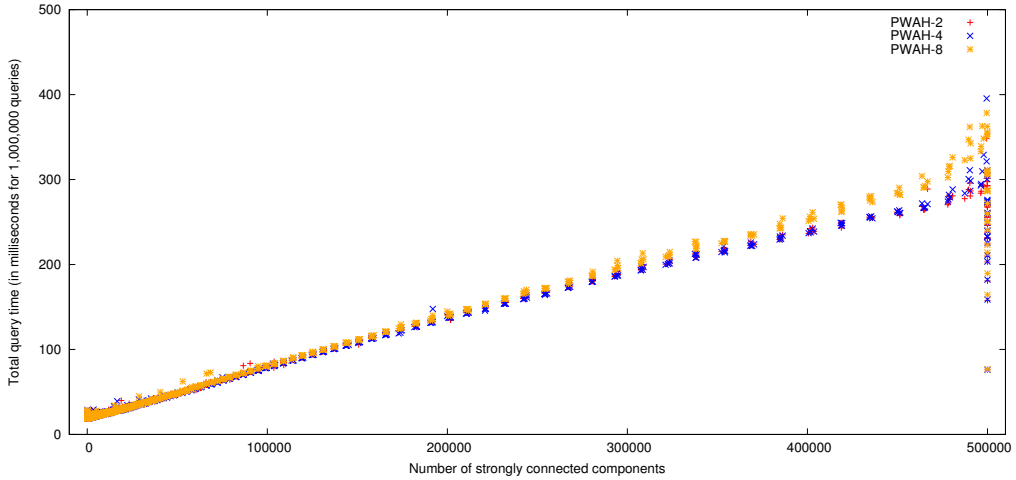


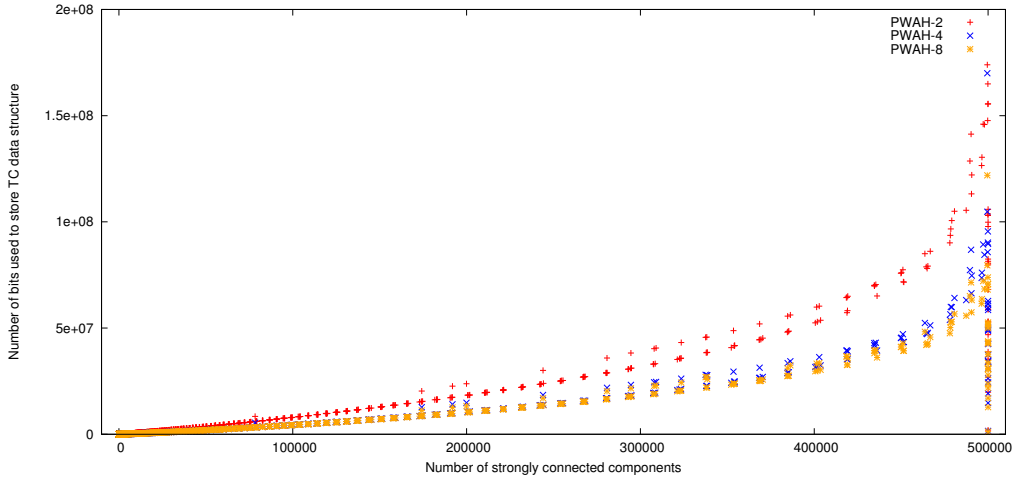
Figure 4.5: Total amount of bits required to store the transitive closure data structure (using PWAH-2 compression) with respect to the number of strongly connected components in random graphs with 500,000 vertices and up to 5,000,000 edges. The estimated number of bits an equivalent reachability matrix would use is illustrated by the continuous blue plot line. Note that both plots contain the same data points but the Y axes are scaled differently.



(a)



(b)



(c)

Figure 4.6: Comparison of PWAH schemes based on (a) construction time, (b) query time and (c) memory usage

near future. When performing a linear scan through the words of a PWAH compressed bit vector, this caching strategy turns out to perform remarkably well. However, when introducing an index (which actually consists of three separate integer arrays) on the PWAH vector, multiple random memory locations have to be read and processed. This effectively renders the operating system's approach to caching useless, deteriorating the lookup performance of the bit vector. Note that the more fine grained indices turn out to be more sensitive to the failing cache, since these take up a significantly larger amount of memory (Figure 4.7c).

4.5 PWAH vs. interval lists, Path-Tree and 3-Hop

4.5.1 Introduction

The correlation between input sizes of random graphs and PWAH performance has been evaluated, but in real life one does scarcely encounter graphs of which the topology resembles that of a random graph. Therefore, the performance of PWAH is evaluated on real life graphs as well, by comparing its performance to that of three other approaches:

1. Nuutila's algorithm with interval lists;
2. Path-Tree [19]: both PTree-1 (optimal path-decomposition by min-cost flow) and PTree-2 (suboptimal path-decomposition);
3. 3-Hop: the 3-Hop Contour approach as described by Jin *et al.* in [18].

For this purpose, a large data set of graphs has been composed, of which the characteristics are listed in Table 4.1 on page 37. These graphs have been split up into three categories which are grouped into single bar charts:

- Graphs from [18, 19], as presented on SIGMOD 2008 and 2009;
- Graphs used for static program analysis;
- Graphs describing the structure of Wikipedia and a road network graph[1].

Unfortunately, it turned out that 3-Hop was not capable of processing most of the input graphs from our data set. Therefore, the comparison of PWAH, interval lists and 3-Hop has been performed based on the graphs used by Jin *et al.* in [18]. The results are presented in Section 4.5.5 on page 47.

Note that some charts are lacking bars from the PTree-1 and PTree-2 algorithms. This is due to either a experiment timeout (5 runs within 3 minutes) or due to a *segmentation fault* which occurred whilst processing the graphs.

Another important thing to note is the scale of the Y axes of the plots in this section: all Y axes are *logarithmically* scaled. Although small differences in timing or memory usage might look bigger, it turned out to be the only way to illustrate the performance of the different approaches to transitive closure computation in a single bar chart.

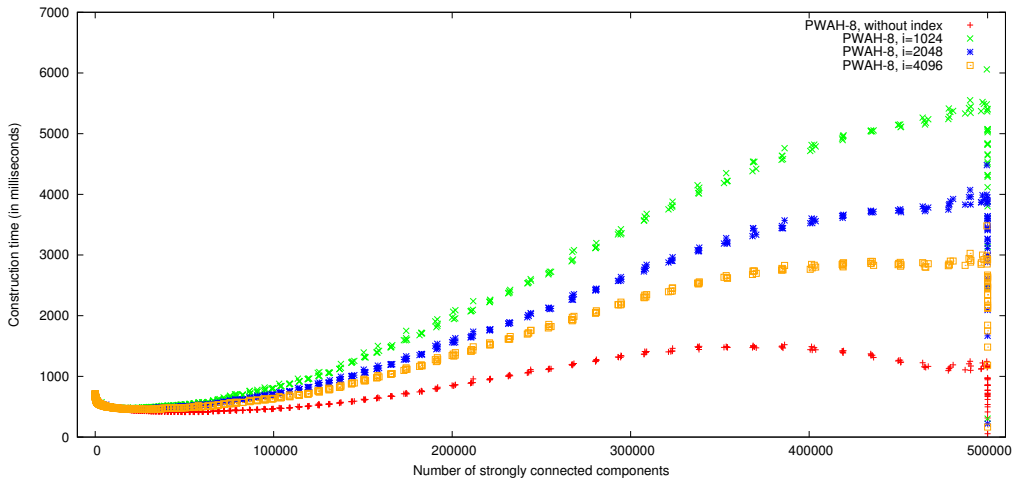
In general, each input graph has been processed five times by each algorithm after which both the construction times and query times were averaged. Since all algorithms are deterministic, the memory usage does not vary over different runs.

4.5.2 Construction time

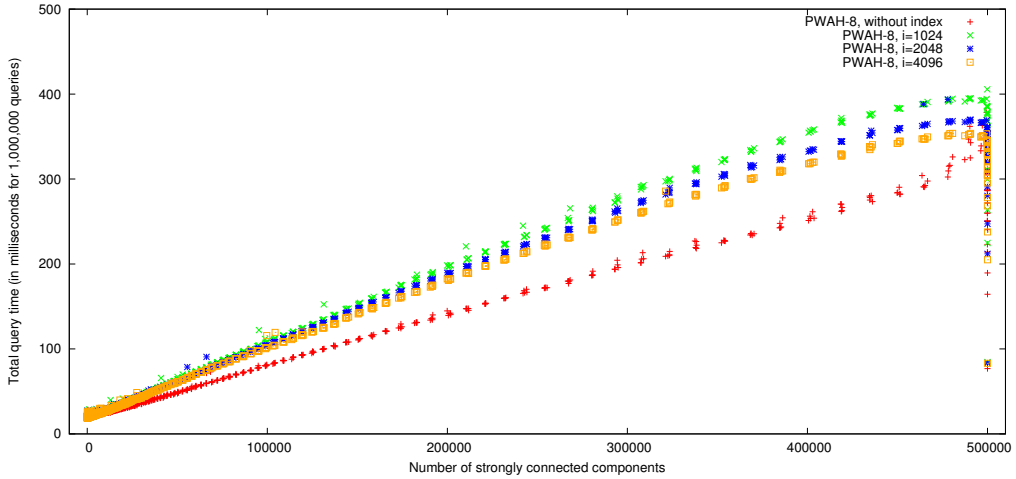
Figure 4.8 on page 48 provides a comparison of construction times (in milliseconds). The difference between the performance of Nuutila's approach (using PWAH bit vectors or interval lists as backend) and the performance of the PathTree algorithms is astounding: around two orders of magnitude. In most cases, interval lists turn out to perform slightly better than the PWAH compressed bit vectors, although the timings are extremely close.

Note that in nine cases, the PathTree implementations were not capable of processing the input graph. Most of the times due to a timeout, caused by heavy *memory swapping*, sometimes by an

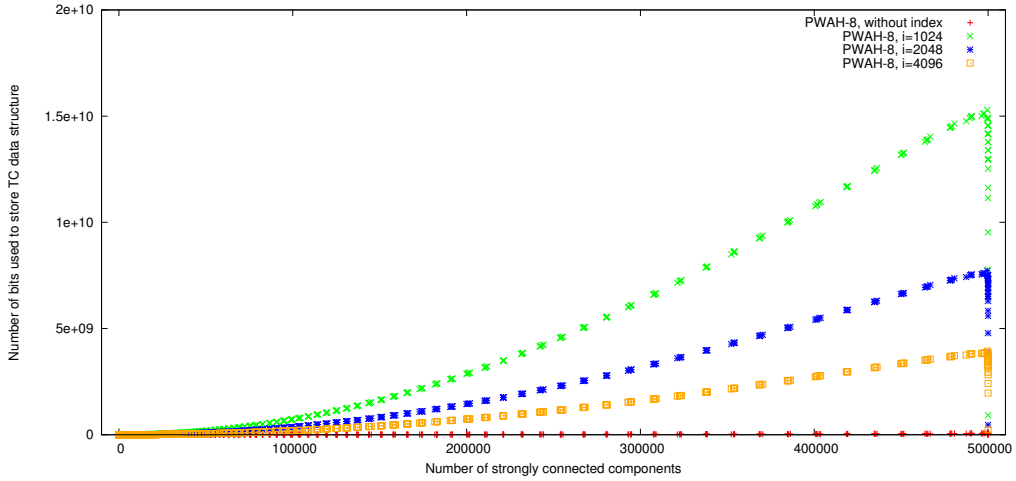
[1] The road network graph is strictly proprietary and was provided by an anonymous third party



(a)



(b)



(c)

Figure 4.7: Plots showing the impact of indexing on construction time, query time and memory usage of PWAH-8

unexplainable *segmentation fault*. Three graphs which are listed in Table 4.1 on page 37 are not depicted in Figure 4.8 since the graphs turned out to bee too small to yield timings significant enough for comparison.

4.5.3 Query time

The time required to perform 1,000,000 random queries on the transitive closure data structures is depicted in Figure 4.9 on page 49. Once more, note the logarithmically scaled Y axes of the plots.

As was the case with comparing construction time, interval lists perform remarkably well. In general, differences in query time are not very large. In spite of their significantly longer construction time, the PTree algorithms do generally not outperform the PWAH and interval approaches in terms of query time.

4.5.4 Memory usage

Figure 4.10 on page 50 contains three plots depicting the memory usage of the transitive closure data structures produced by the different algorithms. These experimental results confirm the theory presented in Section 3.3.7: PWAH-8 will virtually always outperform interval lists in terms of memory usage. In fact, PWAH-8 outperforms PathTree as well, despite its significantly longer construction time.

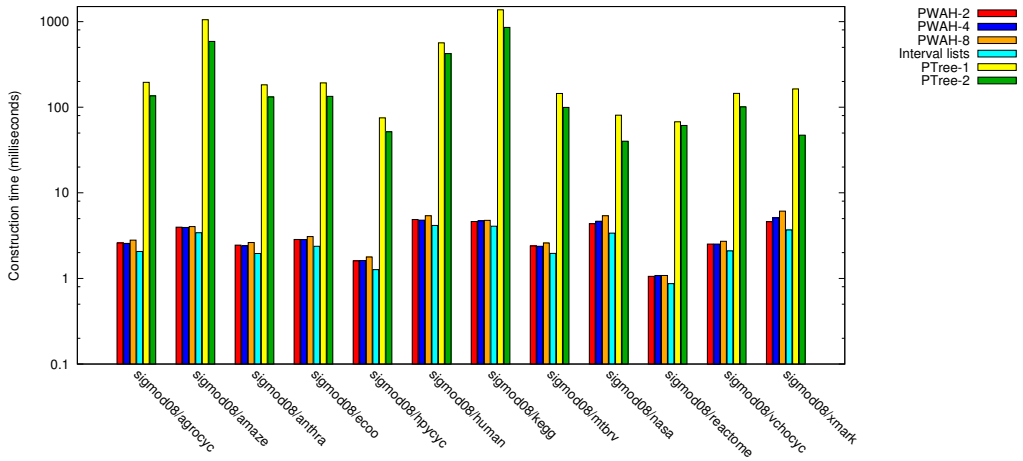
4.5.5 PWAH, Interval lists and 3HOP

The five graphs used by Jin *et al.* in [18] were used to compare 3-Hop to interval lists and PWAH-8. The results are depicted in Figure 4.11 on page 51. As listed in Table 4.1 on page 37, the graphs are acyclic and significantly more dense than most other graphs.

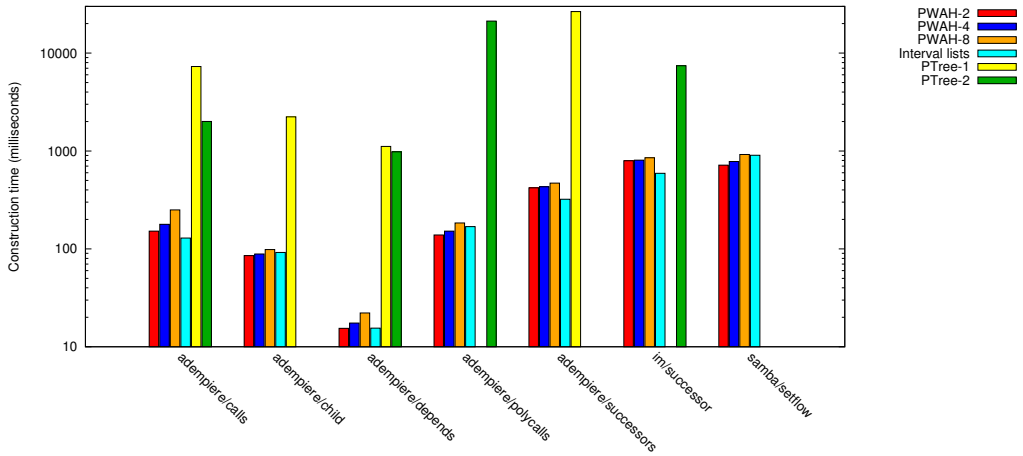
The results shed some light on the reason for the authors of [18] not to include an experimental analysis of the construction time. For one of the graphs, it took the 3-Hop algorithm over 15 minutes to construct the transitive closure data structure. The construction time of the transitive closure of the other graphs varies from ten seconds to 1.5 minutes. The data structure based on interval lists can be constructed a few milliseconds to 50 milliseconds faster than the PWAH-8 structures. The difference with 3-Hop is more significant: *up to four orders of magnitude*.

When looking at the query time, interval lists perform remarkably well: up to one order of magnitude better than PWAH-8, up to two orders of magnitude better than 3-Hop. This is most probably caused by the fact that query time on a PWAH-8 bit vector is linear in the length of the vector, whilst an interval list can be queried within logarithmic time. As listed in Table 4.1 on page 37, the graphs are considerably dense though acyclic and do therefore not contain any strongly connected components of size > 1. Hence, the bit vectors and interval lists will grow quite large, resulting in a more significant difference in query time performance.

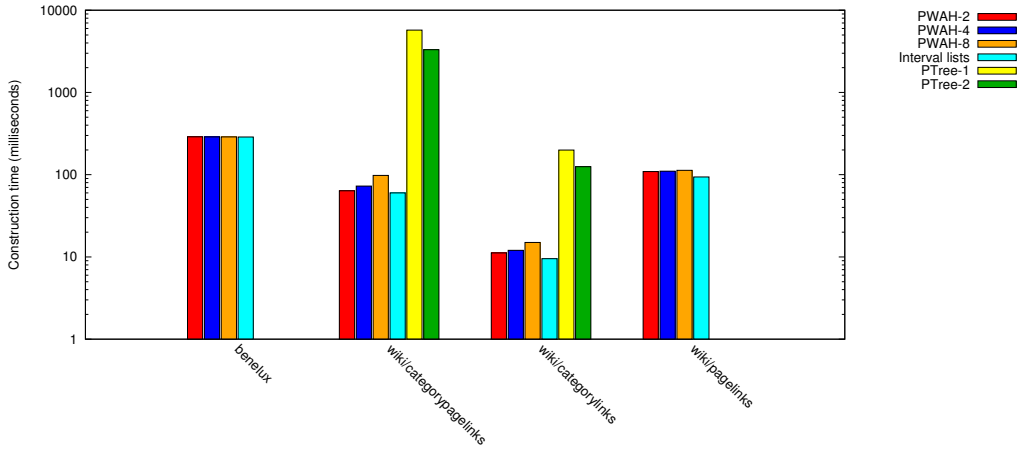
However, when looking at the memory usage, interval lists perform rather dreadful. PWAH-8 is performing best of all three approaches for four out of five graphs, up to little less than one order of magnitude better than 3-Hop and interval lists. As was the case with Path-Tree, in spite of a significantly longer construction time, 3-Hop does generally not outperform PWAH and interval lists in terms of both query time and memory usage.



(a) Construction time of TC of graphs from [19]

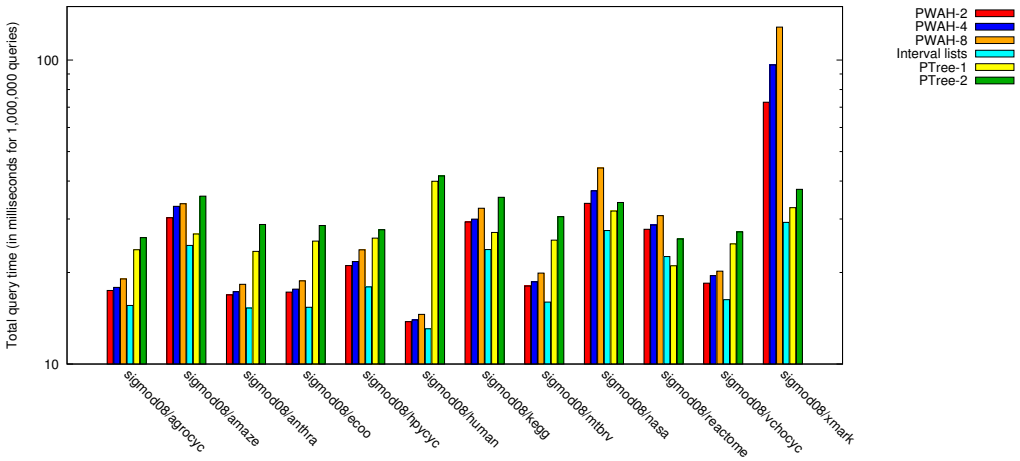


(b) Construction time of TC of graphs

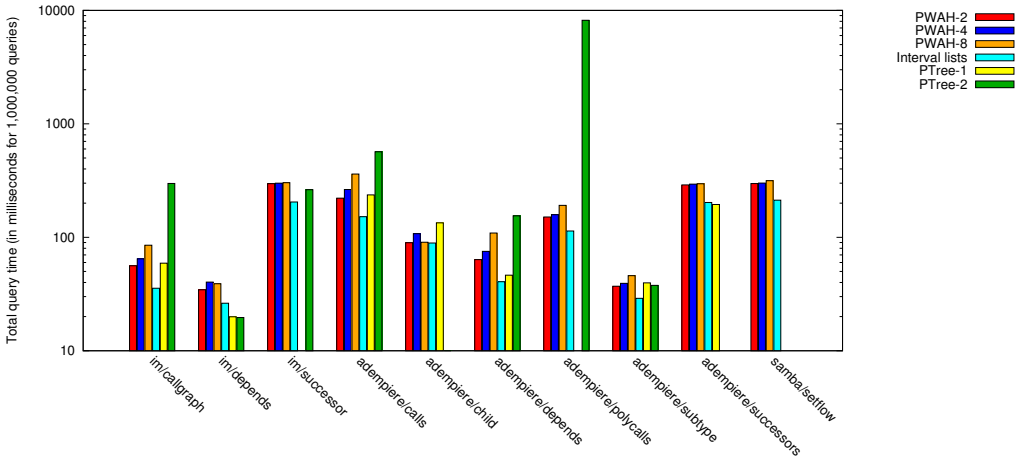


(c) Construction time of TC of graphs

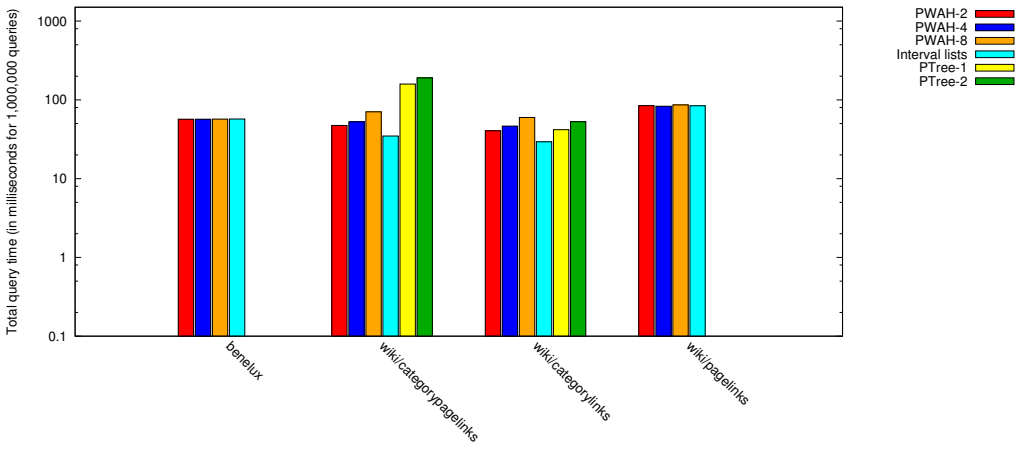
Figure 4.8: Comparison of construction time using multiple real-world graphs



(a) Query time on TC of graphs from [19]

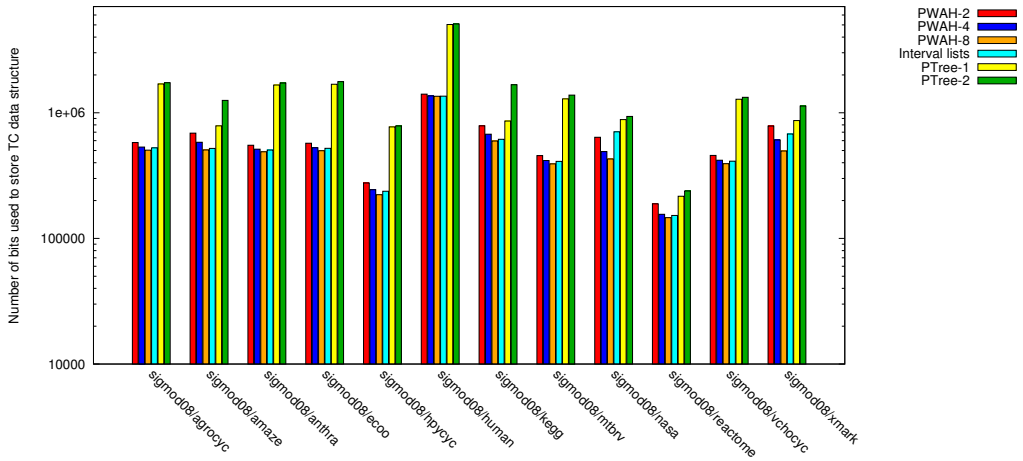


(b) Query time on TC of graphs

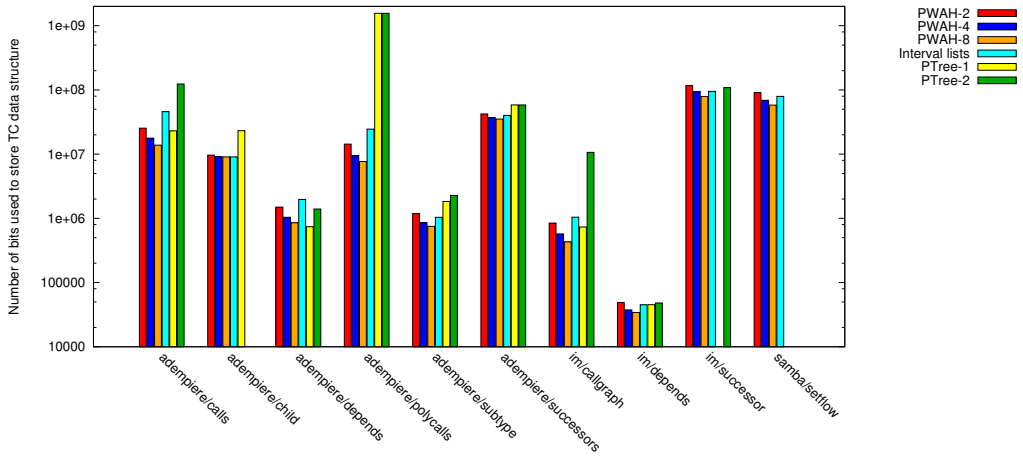


(c) Query time on TC of graphs

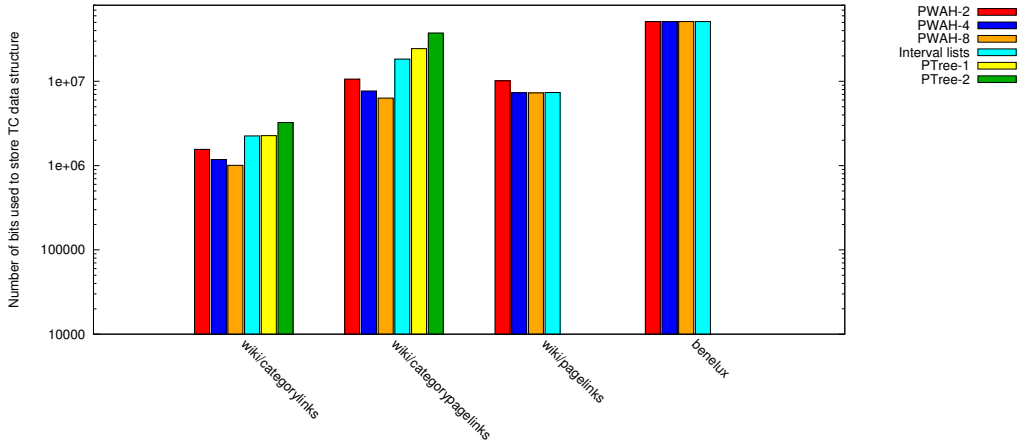
Figure 4.9: Comparison of memory usage using multiple real-world graphs



(a) Memory usage of TC of graphs from [19]

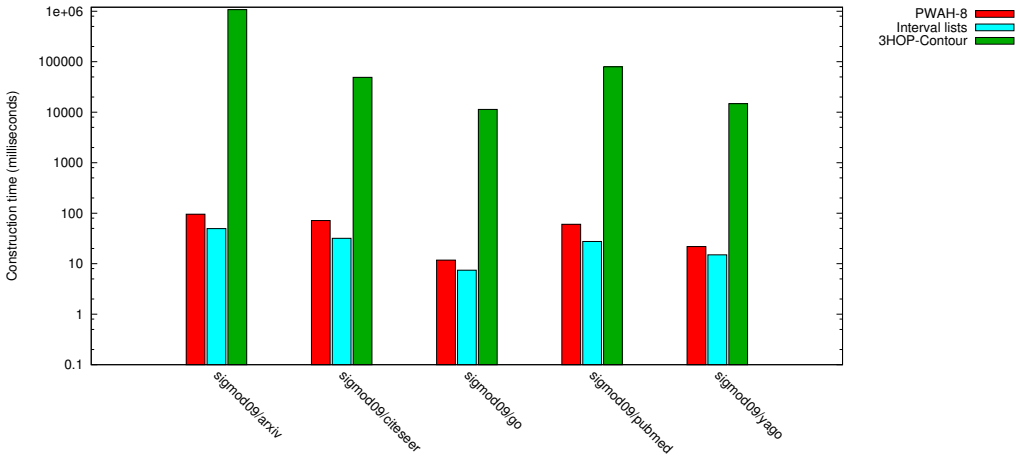


(b) Memory usage of TC of graphs

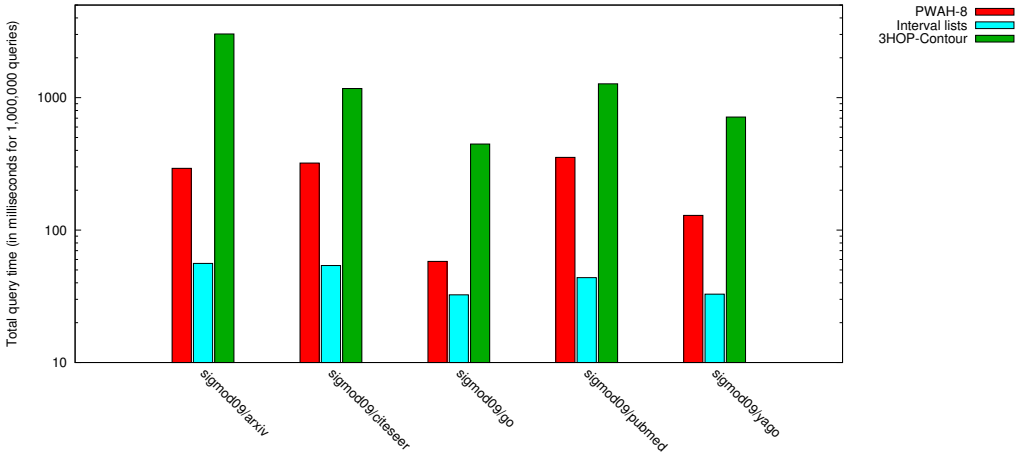


(c) Memory usage of TC of graphs

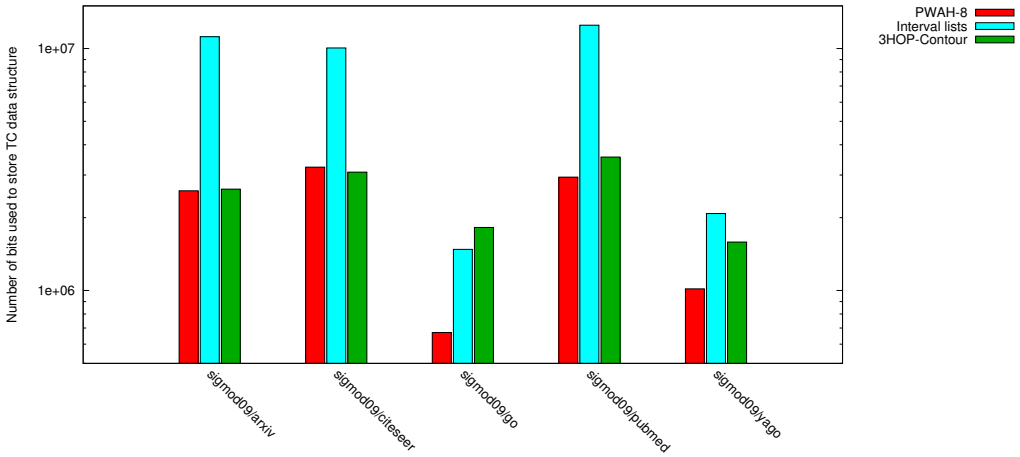
Figure 4.10: Comparison of memory usage using multiple real-world graphs



(a) Comparison of construction time



(b) Comparison of query time



(c) Comparison of memory usage

Figure 4.11: Performance comparison of PWAH-8, interval lists and 3-Hop using graphs from [18]

Chapter 5

Conclusion

5.1 Experimental evaluation

5.1.1 PWAH

Although it was shown that there exists a strong correlation between the number of vertices and the number of edges on one side and the performance of different compression schemes on the other, it is clear that the graph topology has a very significant (maybe even stronger) influence. All of the approaches especially depend on the number of strongly connected components and edges in between them.

Quite surprisingly, indexing PWAH does not yield a performance improvement in terms of query time. It rather results in a deterioration of performance, most likely caused by cache misses.

As expected, the PWAH schemes turn out to be a very competitive opponent to interval lists, PathTree and 3-Hop on all performance tests. Especially with respect to memory usage, PWAH-8 turns out to be highly effective for compressing reachability information, whilst maintaining impressive construction and query times.

5.1.2 Interval lists

As suggested by Nuutila in [24], interval lists are a very interesting option to store reachability information. The lists perform quite impressive in terms of construction time, query time and memory usage. However, PWAH-8 will perform better than interval lists in terms of memory usage in virtually all cases, slightly at expense of construction time and query time performance.

5.1.3 Path-Tree and 3-Hop

Both the Path-Tree [19] and 3-Hop [18] approaches are supported by a very sound theoretical foundation, but turn out to be both slower and use more memory than PWAH and interval lists on virtually all graphs. Although [19] is titled *"Efficiently answering reachability queries on very large directed graphs"*, the implementation turns out to be incapable of processing most large graphs in our data set. Furthermore, the $\mathcal{O}(1)$ query time mentioned in [19] is quite misleading. The theory of the 3-Hop algorithm sounds very interesting, but unfortunately the construction time of the transitive closure renders this approach practically unusable.

5.2 Further research

As is fairly common when conducting a research project, there are some aspects which have not yet been thoroughly investigated, but which might nevertheless be interesting to look at. A few suggestions are listed below.

Multiple queries: sort and iterate

As mentioned in Chapter 3, the largest downside to using compressed bit vectors is the increased query time. In contrast to regular bit vectors, compressed bit vectors do not store information at a fixed offset, hence a full scan over the vector might be required to determine the value of the requested bit.

However, when a sequence of queries is to be answered by the compressed bit vector, it might be beneficial to sort the queries first (in order of increasing bit index) and use an iterator to determine the values of the queried bits. This approach would require only one scan over the bit vector, yielding an expected run time of $\mathcal{O}(k\ log_2(k) + n)$ (for k queries on a bit vector of size n, assuming sorting can be done in $\mathcal{O}(k\ log_2(k))$ time.

Optimised interval lists

Throughout this thesis, interval lists were assumed to use two 32-bit integers to store a single interval. If the maximum range of the values of the interval list is known beforehand, it is possible to reduce the number of bits used to store a single interval. It might be interesting to perform an analytical and experimental comparison between PWAH-8 and the optimised interval lists.

Matrix multiplication on sparse matrices

The concept of using matrix multiplication for transitive closure computation has been described in Section 2.4. Matrix multiplication and matrix compression (primarily for sparse/dense matrices) are subjects of ongoing research [44], it might therefore be interesting to conduct an experiment to compare different approaches to multiplication of compressed matrices for transitive closure computation.

Other types of compression

This thesis introduced the Partitioned Word Aligned Hybrid compression scheme, based on WAH by Wu *et al.* [42]. It has been shown that reachability information obtained by performing a depth first search is structured in a way very suitable for compression. There exist numerous types of compression schemes, each having its own advantages and disadvantages. Looking into other compression schemes for compressing reachability information could yield interesting results.

More detailed theoretical analysis

It turned out to be very hard to actually *prove* certain properties of the behaviour of PWAH. Considering the interesting experimental results, a more in-depth study of the theory behind PWAH might improve our understanding of (the boundaries of) its performance.

Introduce PWAH in other fields of research

The concept of bit vector compression is being used in many fields of research and industry. Especially in the situations in which WAH [42] is currently being employed, it should be interesting to consider the usage of PWAH, which provides a more fine-grained compression scheme.

Acknowledgements

The research project on which this thesis is based, as well as the thesis itself would not have been possible without the people mentioned below. I am very grateful to all of them for providing me with the support I needed to conduct this research project.

Oege de Moor has given me the opportunity to stay in his research group at the University of Oxford for the entire length of the project. His valuable advice, endless optimism and continuous encouragements have been of inexpressible importance to the project and the brilliant time I had in Oxford whilst working on transitive closures. Furthermore it is him I need to thank for introducing me to other scientists within the department, which lead to acceptance for a D.Phil. (PhD) in Computer Science at the University of Oxford.

Although **Arno Siebes** did not concern himself too much with the details of my research, he has been a very supportive supervisor, even though he was based in Utrecht. His advice regarding personal affairs – both during the research project and in the years before that – has been of invaluable importance and eventually helped me decide to pursue a career in academia.

My first serious encounter with research would not have been such a success if it were not for the presence of **Max Schäfer** in office 002 of the University of Oxford Computing Laboratory. His humour and irony have been of great help during difficult times. Discussions about research, life, the universe (basically everything) have been a very welcome addition to the daily routines of a research student. Also, I feel obliged to thank him for teaching me the ins and outs of refactorings, which made me decide to not ever even consider pursuing a career in that field. Last, but not least he has provided me with an incredible amount of useful comments after having proofread this thesis, for which I am very thankful.

Steven Woudenberg has been a great friend even though I had left Utrecht, bringing me the right things at the right time. A single phone call with him could made me feel like I had never left my home city. Furthermore, he has been a great host for the times I briefly returned to Utrecht and his visits to Oxford were always very cheerful.

Last but not least, **my parents** deserve the biggest credit of all. Although they did not understand all of the choices I made in the past few years, they have always endeavoured to support me and have always respected my decisions. It is that unconditional support which has been of vital importance to all of my achievements.

Furthermore, there are a number of people I would like to thank for being a great friend and for providing me with input and feedback on my thoughts: Anne Leewis, Claire Lowdon, Frank Tip, Julian Tibble, Marie-Jette Wierbos, Michelle Meekes, Neelam Hassanali, Pavel Avgustinov, Rianne 't Hoen, Sander van der Waal and Silvia Breu.

Appendix A

Tarjan's algorithm: example

The following step-by-step example illustrate how Tarjan's algorithm processes the graph depicted in figure 2.1 on page 11. The first steps are also listed in section 2.2.3 on page 10.

1. DfsVisit(a) – Visit vertex a:
   (a) Mark a as CCR of itself: $CCR[a] = a$, store DFS sequence number 1 for a: $D[a] = 1$
   (b) Iterate over adjacent vertices: DfsVisit(b)

2. DfsVisit(b) – Visit vertex b:
   (a) Mark b as CCR of itself: $CCR[b] = b$, store DFS sequence number 2 for b: $D[b] = 2$
   (b) Iterate over adjacent vertices:
      i. Adjacent vertex a has been visited before, not calling DfsVisit
      $C[a] = Nil$ and $D[CCR[a]] < D[CCR[b]]$ $(1 < 2)$,
      therefore b inherits the CCR of a: $CCR[b] \leftarrow CCR[a]$ (thus, $CCR[b] = a$)
      ii. Adjacent vertex c has not been visited before: DfsVisit(c)

3. DfsVisit(c) – Visit vertex c:
   (a) $CCR[c] = c$, $D[c] = 3$
   (b) Iterate over adjacent vertices:
      i. Adjacent vertex b has been visited before, not calling DfsVisit
      $C[b] = Nil$ and $D[CCR[b]] < D[CCR[c]]$ $(1 < 3)$,
      therefore c inherits the CCR of b: $CCR[c] \leftarrow CCR[b]$ (thus, $CCR[c] = a$)
      ii. Adjacent vertex d has not been visited before: DfsVisit(d)

4. DfsVisit(d) – Visit vertex d:
   (a) $CCR[d] = d$, $D[d] = 4$
   (b) Iterate over adjacent vertices: DfsVisit(e)

5. DfsVisit(e) – Visit vertex e:
   (a) $CCR[e] = e$, $D[e] = 5$
   (b) Iterate over adjacent vertices:
      i. Adjacent vertex d has been visited before, not calling DfsVisit
      $C[d] = Nil$ and $D[CCR[d]] < D[CCR[e]]$ $(4 < 5)$,
      therefore e inherits the CCR of d: $CCR[e] \leftarrow CCR[d]$ (thus, $CCR[e] = d$)
   (c) No more adjacent vertices. $CCR[e] = d(\neq e)$, therefore push e on stack S: $S = \{e\}$
   (d) Return DFS call to d

6. Continue processing at vertex d
   (c) No more adjacent vertices left. $CCR[d] = d$, therefore d becomes *final* component root

   i. Create a new component: $C[d] = new\ component$
   ii. Stack $S = \{e\}$, pop component vertices off the stack:
   Pop $e \rightarrow C[e] = C[d]$
   $S = \{\}$

(d) Return DFS call to c

7. Continue processing at vertex c
   (c) No more adjacent vertices left. $CCR[c] = a(\neq c)$, therefore push c on stack S: $S = \{c\}$
   (d) Return DFS call to b

8. Continue processing at vertex b
   (c) No more adjacent vertices left. $CCR[b] = a(\neq b)$, therefore push b on stack S: $S = \{c, b\}$
   (d) Return DFS call to a

9. Continue processing at vertex a
   (b) Continue iterating over adjacent vertices: DfsVisit(f)

10. DfsVisit(f)
   (a) $CCR[f] = f$, $D[f] = 6$
   (b) Iterate over adjacent vertices: DfsVisit(g)

11. DfsVisit(g)
   (a) $CCR[g] = g$, $D[g] = 7$
   (b) Iterate over adjacent vertices:
      i. Adjacent vertex d has been visited before, not calling DfsVisit
      $C[d] \neq Nil \rightarrow g$ does not reside within d's SCC $\rightarrow CCR[g]$ remains unchanged.
      ii. Adjacent vertex f has been visited before, not calling DfsVisit
      $C[f] = Nil$ and $D[CCR[f]] < D[CCR[g]]$ $(6 < 7)$,
      therefore g inherits the CCR of f: $CCR[g] \leftarrow CCR[f]$ (thus, $CCR[g] = f$)
   (c) No more adjacent vertices left. $CCR[g] = f(\neq g)$, therefore push g on stack S: $S = \{c, b, g\}$
   (d) Return DFS call to f

12. Continue processing at vertex f
   (c) No more adjacent vertices left. $CCR[f] = f$, therefore f becomes *final* component root
      i. Create a new component: $C[f] = new\ component$
      ii. Stack $S = \{c, b, g\}$, pop component vertices off the stack:
      Pop $g \rightarrow C[g] = C[f]$
      $S = \{c, b\}$
   (d) Return DFS call to a

13. Continue processing at vertex a
   (b) Continue iterating over adjacent vertices: DfsVisit(h)

14. DfsVisit(h)
   (a) $CCR[h] = h$, $D[h] = 8$
   (b) Iterate over adjacent vertices: DfsVisit(i)

15. DfsVisit(i)
   (a) $CCR[i] = i$, $D[i] = 9$
   (b) Iterate over adjacent vertices:
      i. Adjacent vertex c has been visited before, not calling DfsVisit
      $C[c] = Nil$ and $D[CCR[c]] < D[CCR[i]]$ $(3 < 9)$,
      therefore i inherits the CCR of c: $CCR[i] \leftarrow CCR[c]$ (thus, $CCR[i] = a$)

ii. Adjacent vertex e has been visited before, not calling DfsVisit
$C[e] \neq Nil \rightarrow i$ does not reside within e's SCC $\rightarrow CCR[i]$ remains unchanged.
iii. Adjacent vertex h has been visited before, not calling DfsVisit
$C[h] = Nil$ but $D[CCR[h]] \not< D[CCR[i]]$ $(8 \not< 3)$.
Hence, i *does* reside within h's SCC. However, i's CCR ($CCR[i] = a$) is considered more optimal than h's CCR ($CCR[h] = h, D[a] < D[h]$). Therefore i *does not* inherit the CCR of h.
iv. Adjacent vertex j has not been visited before: DfsVisit(j)

16. DfsVisit(j)
    (a) $CCR[j] = j$, $D[j] = 10$
    (b) Iterate over adjacent vertices: none
    (c) No more adjacent vertices left. $CCR[j] = j$, therefore j becomes *final* component root
        i. Create a new component: $C[j] = new\ component$
        ii. Stack $S = \{c, b\}$, $D[b] < D[i] \rightarrow$ no vertices to be popped off the stack
    (d) Return DFS call to i

17. Continue processing at vertex i
    (c) No more adjacent vertices left. $CCR[b] = a \neq b$, therefore push i on stack S: $S = \{c, b, i\}$
    (d) Return DFS call to h

18. Continue processing at vertex h
    (c) Adjacent vertex i has been processed, $CCR[i]$ changed. Since $C[i] = Nil$, i belongs to the same component as h and h needs to update its CCR: $CCR[h] \leftarrow CCR[i]$ (thus, $CCR[h] = a$).
    (d) No more adjacent vertices left. $CCR[h] = a \neq h$, therefore push h on stack S: $S = \{c, b, i, h\}$
    (e) Return DFS call to a

19. Continue processing at vertex a
    (c) No more adjacent vertices left. $CCR[a] = a$, therefore a becomes *final* component root
        i. Create a new component: $C[a] = new\ component$
        ii. Stack $S = \{c, b, i, h\}$, pop component vertices off the stack:
        Pop $h \rightarrow C[h] = C[a]$
        Pop $i \rightarrow C[i] = C[a]$
        Pop $b \rightarrow C[b] = C[a]$
        Pop $c \rightarrow C[c] = C[a]$
        $S = \{\}$
    (d) Return DFS call, algorithm terminates

Appendix B

Code details

Introduction

The research project which formed the foundation of this thesis started out using only the Java programming language. However, in order to be able to compare the run time of the implementation using compressed bit vectors to Path-Tree [19] and 3-Hop [18], using Java was no longer considered an option.

Henceforth, C++ was used as new default language and most code was rewritten in C++. Specific parts of the code are only available in one of the languages, like an implementation of PWAH (C++ only) and the graph generator (Java only).

All code is available on request: `Sebastiaan.van.Schaik@comlab.ox.ac.uk`.

Java code

The Java code contains implementations of:

- A general directed graph (write once, read many), including parser for multiple file formats
- Graph exporting functionality (to multiple file formats)
- Tarjan's algorithm
- Simple matrix multiplication for transitive closure computation
- BFS and DFS algorithms for transitive closure computation
- Interval lists
- A random graph generator

C++ code

The final C++ project contained a total of around 4,500 lines of C++ code, according to *SLOCCount* [41]. It does not have any external dependencies, except for standard C/C++ library classes and functions. The C++ code contains implementations of:

- A general directed graph (write once, read many), including parser
- Tarjan's algorithm
- The WAH compression scheme, including iterators
- The PWAH compression schemes, including iterators
- Interval lists

Bibliography

[1] R. Acharya1, A. Kumar, P.S. Bhat, C. M. Lim, S. S. Iyengar, N. Kannathal, and S.M. Krishnan. Classification of cardiac abnormalities using heart rate signals. *Medical and Biological Engineering and Computing*, 42(3):288–293, 2004.

[2] ADempiere Community. ADempiere. `http://www.adempiere.org`.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.

[4] Anita Burgun and Olivier Bodenreider. An ontology of chemical entities helps identify dependence relations among gene ontology terms. In *Proceedings of First Symposium on Semantic Mining in Biomedicine*, 2005.

[5] Edmund M. Clarke. The birth of model checking. In *25 Years of Model Checking*, pages 1–26, 2008.

[6] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, second edition, 2001.

[8] John Cristy and ImageMagick Studio LLC. ImageMagick. `http://www.imagemagick.org`.

[9] Souripriya Das, Eugene Inseok Chong, George Eadon, and Jaannathan Srinivasan. Supporting ontology-based semantic matching in rdbms. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1054–1065. VLDB Endowment, 2004.

[10] EcoCyc developers and contributors. EcoCyc. `http://www.ecocyc.org`.

[11] Michael J. Fischer and Albert R. Meyer. Boolean matrix multiplication and transitive closure. In *FOCS*, pages 129–131. IEEE, 1971.

[12] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[13] Sumit Ganguly, Ravi Krishnamurthy, and Abraham Silberschatz. An analysis technique for transitive closure algorithms: A statistical approach. In *Proceedings of the Seventh International Conference on Data Engineering*, pages 728–735, Washington, DC, USA, 1991. IEEE Computer Society.

[14] Alan Gibbons, Aris Pagourtzis, Igor Potapov, and Wojciech Rytter. Coarse-grained parallel transitive closure algorithm: Path decomposition technique. *Computer Journal*, 46(4):391–400, 2003.

[15] Lee Giles, Isaac Councill, and the CiteSeer Project. CiteSeer. `http://citeseerx.ist.psu.edu`.

[16] A. V. Goldberg, É. Tardos, and R. E. Tarjan. Network Flow Algorithms. In B. Korte, L. Lovász, H. J. Prömel, and A. Schrijver, editors, *Flows, Paths, and VLSI Layout*, pages 101–164. Springer Verlag, 1990.

[17] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, 1990.

[18] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-hop: a high-compression indexing scheme for reachability query. In *SIGMOD '09: Proceedings of the 35th SIGMOD international conference on Management of data*, pages 813–826, New York, NY, USA, 2009. ACM.

[19] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 595–608, New York, NY, USA, 2008. ACM.

[20] Vincent Lacroix, Cristina G. Fern, and Marie-france Sagot. Reaction motifs in metabolic networks. In *In Proceedings of the 5th international Workshop on Algorithms in BioInformatics (WABI)*, pages 178–191, 2005.

[21] Enrico Macii. A discussion on explicit methods for transitive closure computation based on matrix multiplication. *Asilomar Conference on Signals, Systems and Computers*, 0:799, 1995.

[22] Mark E. J. Newman. Random graphs as models of networks. Technical report, 2002.

[23] Mark E. J. Newman. Random graphs with clustering. 2009.

[24] Esko Nuutila. *Efficient Transitive Closure Computation in Large Digraphs*. PhD thesis, Finnish Academy of Technology, 1995. http://www.cs.hut.fi/ enu/tc.html.

[25] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, third edition, September 2007.

[26] Thomas Reps. Program analysis via graph reachability. In *ILPS '97: Proceedings of the 1997 international symposium on Logic programming*, pages 5–19, Cambridge, MA, USA, 1997. MIT Press.

[27] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, New York, NY, USA, 1995. ACM.

[28] Sara Robinson. Toward an optimal algorithm for matrix multiplication. *SIAM News*, 38(9):1–5, November 2005.

[29] B. Roy. Transitivité et connexité. *Comptes Rendus de l'Académie des Sciences Paris*, 249:216–218, 1958.

[30] Klaus Simon. An improved algorithm for transitive closure on acyclic digraphs. In Laurent Kott, editor, *ICALP*, volume 226 of *Lecture Notes in Computer Science*, pages 376–386. Springer, 1986.

[31] Kurt Stockinger, Dirk Duellmann, Wolfgang Hoschek, and Erich Schikuta. Improving the performance of high energy physics analysis through bitmap indices. In *11th International Conference on Database and Expert Systems Applications, Greenwich, London*, pages 835–845. Springer-Verlag, 2000.

[32] Volker Strassen. The asymptotic spectrum of tensors and the exponent of matrix multiplication. In *FOCS*, pages 49–54. IEEE, 1986.

[33] Fabian Suchanek, Gjergji Kasneci, Garhard Weikum, Johannes Hooffart, and Edwin Lewis-Kelham. The YAGO Project. `http://www.mpi-inf.mpg.de/yago-naga/yago/`.

[34] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[35] The Gene Ontology Consortium. The Gene Ontology Project. `http://www.geneontology.org`.

[36] Andrew Tridgell and the Samba Team. Samba. `http://www.samba.org`.

[37] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD Conference*, pages 845–856, 2007.

[38] Cornell University. arXiv. `http://www.arxiv.org`.

[39] U.S. National Institutes of Health, National Center for Biotechnology Information, National Library of Medicine. PubMed Central. `http://www.pubmedcentral.nih.gov`.

[40] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9:11–12, 1962.

[41] David Wheeler. SLOCCount. `http://www.dwheeler.com/sloccount/`.

[42] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. An efficient compression scheme for bitmap indices. Technical report, ACM Transactions on Database Systems, 2004.

[43] Ming-Chuan Wu and Alejandro P. Buchmann. Encoded bitmap indexing for data warehouses. In *ICDE*, pages 220–230. IEEE Computer Society, 1998.

[44] Raphael Yuster and Uri Zwick. Fast sparse matrix multiplication. *ACM Transactions on Algorithms*, 1(1):2–13, 2005.