A Memory Efficient Reachability Data Structure Through Bit Vector Compression

Sebastiaan J. van Schaik[*] and Oege de Moor[†]
University of Oxford Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK
{Sebastiaan.van.Schaik,Oege.de.Moor}@comlab.ox.ac.uk

ABSTRACT

When answering many reachability queries on a large graph, the principal challenge is to represent the transitive closure of the graph compactly, while still allowing fast membership tests on that transitive closure. Recent attempts to address this problem are complex data structures and algorithms such as Path-Tree and 3-HOP. We propose a simple alternative based on a novel form of bit-vector compression. Our starting point is the observation that when computing the transitive closure, reachable vertices tend to cluster together. We adapt the well-known scheme of word-aligned hybrid compression (WAH) to work more efficiently by introducing word partitions. We prove that the resulting scheme leads to a more compact data structure than its closest competitor, namely interval lists. In extensive and detailed experiments, this is confirmed in practice. We also demonstrate that the new technique can handle much larger graphs than alternative algorithms.

Categories and Subject Descriptors

E.1 [**Data**]: Data structures—*Graphs and networks*; E.2 [**Data**]: Data storage representations; E.4 [**Data**]: Coding and information theory—*Data compaction and compression*

General Terms

Algorithms, performance

Keywords

Transitive closure, graph indexing, reachability queries, bit vector compression, WAH, PWAH.

1. INTRODUCTION

The problem of determining – given a directed graph – whether one vertex can be reached via a sequence of edges from another vertex, is ubiquitous. It arises, for example, in computational biology, analysis of social networks, model checking and route planning. Our own interest in the problem stems from applications in program analysis [15, 14], for the purpose of compiler optimisations or to find bugs. It is common, for example, that we need to compute reachability in the graph representing all potential function calls. Other applications within program analysis include points-to analysis and interprocedural dataflow analysis. In all these examples, the graphs in question are sizable, typically having hundreds of thousands or even several million vertices.

If we only wish to compute the result of a handful of reachability queries, clearly the preferred method is to conduct a depth first search from the source vertex, and test whether the target vertex is reached. Each query then takes time and space linear in the size of the graph.

In our applications, there are many reachability queries on a single graph. In these circumstances it is worthwhile to preprocess the graph in order to speed up queries. Formally, computing reachability boils down to computing the transitive closure of the edge relation, so it makes sense to simply pre-compute the entire transitive closure. One way of doing that is the well-known Floyd-Warshall algorithm [20, 16, 5] that computes a boolean matrix representation of transitive closure via dynamic programming. The algorithm itself takes cubic time and quadratic space. Once that time and space has been spent, the reachability queries themselves take only constant time.

It is certainly possible to do better than Floyd-Warshall in terms of asymptotic time complexity, for instance using the equivalence with matrix multiplication. Unfortunately the quadratic space requirements of the Floyd-Warshall algorithm are shared by any explicit representation of transitive closure: given a binary relation R, its transitive closure R^+ may have a size that is quadratic in the size of R. It follows that an explicit representation of the transitive closure is not feasible in practice. What is required, therefore, is a compact representation of the transitive closure that still allows for efficient query time, while avoiding the worst-case quadratic blowup in common cases.

Numerous authors have addressed this challenge before. In a landmark PhD dissertation in 1995 [11], Nuutila showed how to instrument Tarjan's algorithm [19] for computing strongly connected components via depth-first search so that it also computes the transitive closure. He furthermore recommended the use of a simple data structure named *interval lists* to achieve compactness. In a sequence of SIGMOD papers in 2008 [9] and 2009 [8], Jin *et al.* proposed some very

[*]University of Oxford and Utrecht University
[†]University of Oxford

sophisticated data structures to achieve memory efficiency. While their stated goal is to process very large graphs, the size of these graphs is orders of magnitude smaller than those in our applications. Using the implementations that accompany these papers, we found that they often run out of memory on our benchmarks.

Briefly, we achieve better performance as follows. When computing the transitive closure via depth-first search, vertices that are adjacent in a topological sort tend to cluster together in adjacency lists. It is therefore possible to apply run-length data compression to represent these clusters compactly. We introduce a new variant of bit vector compression precisely for that purpose. The result is a new transitive closure algorithm that is much simpler than previous approaches, while yielding vastly superior performance in practice.

In summary, this paper makes the following contributions:

- the use of compressed bit vectors to compactly represent transitive closures;
- a new bit vector compression scheme named Partitioned Word-Aligned Hybrid (PWAH) compression that (for this particular application) significantly outperforms the WAH scheme on which it is based in terms of memory usage;
- an extensive experimental evaluation of different algorithms, on all the benchmarks proposed by Jin *et al.* as well as much larger examples. This includes a comparison between the methods of Jin *et al.* and Nuutila's original algorithm, which has not been previously considered in the literature.

We start with summarising the preliminaries in Section 2. In particular, we present a simple version of Nuutila's algorithm, and deduce the interface of the abstract datatype for representing reachability information. Next, we present our new compression scheme in Section 3. A theoretical analysis is presented in Section 4, proving that PWAH always outperforms interval lists in terms of memory usage. We briefly review previous work on reachability (including Path-Tree [9] and 3-HOP [8]) in Section 5, some of which is used in the experimental evaluation in Section 6. We conclude in Section 7.

2. PRELIMINARIES

This section outlines the basic algorithm we use for computing and storing the transitive closure. It consists of a number of simple steps. First, the problem is reduced to computing the transitive closure of the condensation graph (*i.e.* the graph induced by strongly connected components). Second, we present an algorithm for computing the transitive closure of acyclic graphs. The key data structure needed for that is an implementation of sets with a fast union operation and low memory requirements. We review the implementation of that interface through *interval lists.* Finally, we briefly outline how Nuutila combined all these steps in a single depth-first search of the original input graph.

Condensation Graph. If a graph contains cycles, all vertices in the cycle have equivalent reachability. To wit, write $reach(v, w)$ when there exists a path from v to w. If v and w are on the same cycle, we have

$$\begin{aligned} reach(v,u) &\equiv reach(w,u) \\ reach(u,v) &\equiv reach(u,w) \end{aligned}$$

for all u. It would be wasteful, therefore, to store this same reachability information separately for all vertices in the cycle. Instead, we compute the transitive closure of the *condensation graph.* The condensation graph has as vertices the strongly connected components of the original graph, and there is an edge (c, d) whenever there exists $v \in c$ and $w \in d$ so that (v, w) is an edge in the original graph. Write $reach'(c, d)$ if d is reachable from c in the condensation graph, and write $[v]$ for the strongly connected component of v. We then have

$$reach(v,w) \equiv reach'([v],[w])$$

Note that the condensation graph is acyclic, apart from the possible existence of self-loops.

Reachability in an acyclic graph. There exists a simple way of computing reachability in an acyclic graph. First sort the vertices in topological order, so that all edges point forward in the sequence of vertices. Then work from back to front, computing for each vertex the set of all other vertices that are reachable. The algorithm is shown below:

```
$[v_1, v_2, \ldots, v_n]$ := topSort(V,E)
for i := n downto 1 do
    $reachable[v_i] := \bigcup\{reachable[v_j] \cup \{v_j\} \mid (v_i, v_j) \in E\}$
```

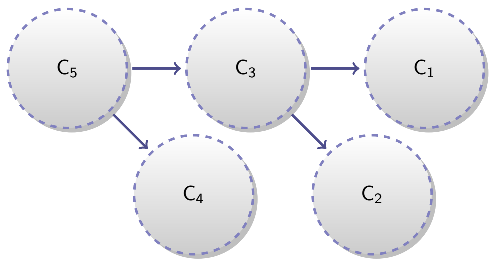
The key insight is that whenever $(v_i, v_j) \in E$, it follows by definition of topological sort that $i < j$, so $reachable[v_j]$ has already been computed by the time we compute $reachable[v_i]$. Furthermore, it is easy to compute the *reach* predicate in terms of the array *reachable*:

$$reach(v,w) \equiv w \in reachable[v]$$

To make this algorithm work on the condensation graph, we need to cater for the possible existence of self-loops. This is easy: do the topological sort with respect to non-loop edges, and initialise the elements of the *reachable* array to empty. With these modifications, $reachable[v_i]$ will contain v_i itself precisely when there is a self-loop $(v_i, v_i) \in E$.

The above algorithm relies on an abstract data type of sets that has the following operations: big union of a collection of sets, binary union, add an element and a test for membership. Furthermore our aim is to ensure that the sets are compactly represented.

Implementation with interval lists. The characteristics of the above algorithm actually suggest a simple way of achieving such a compact representation. Because we process the vertices of the condensation graph in reverse topological order, the elements of the *reachable* array will often consist of nodes that are adjacent in the topological sort. Figure 1 contains a very compact example which provides some intuition on this notion. Consider, for example, the reachability information for C_5. Here we merge the reachable sets of C_3 and C_4, while adding $\{C_3, C_4\}$, resulting in $\{C_1, C_2, C_3, C_4\}$. Doing such merge operations on larger examples evidently often results in the merge of many contiguous subsequences of the topological sort. These contiguous subsequences can be efficiently represented by interval lists.



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	$R(C_5) = R(C_3) \cup R(C_4) \cup \{C_3, C_4\}$

Figure 1: Merging reachability information of strongly connected components

To illustrate, to represent the set of vertices

$$\{v_3, v_5, v_6, v_7, v_8, v_9, v_{15}, v_{16}\}$$

we would use the list of intervals

$$[(v_3, v_3), (v_5, v_9), (v_{15}, v_{16})]$$

As the intervals are presented in order, the membership test takes only logarithmic time. The union operation is a simple merge, which takes time linear in the size of the list (not the set). The other operations are easily derived from this.

Nuutila's algorithm. Observant readers will have noticed that the main two steps of the above algorithm (compute condensation graph, topological sort) are well-known applications of depth-first search [19]. It is natural, therefore, to wonder whether it would not be possible to do all of the above in one depth-first search of the graph. This question was answered in the affirmative by Nuutila in his PhD thesis [11]. His algorithm 'STACK_TC' (which will be referred to as 'Nuutila's algorithm' throughout this paper) does some additional book keeping in the form of extra stacks. The details are not important to the rest of this paper, but we do note that for the experiments in Section 6, we use an implementation of Nuutila's one-pass algorithm.

3. BIT VECTOR COMPRESSION

3.1 Introduction

In the previous section we observed that Nuutila's algorithm yields highly clustered reachability information. This information can be stored in an efficient way using interval lists, but basically every data structure suitable for compressing contiguous sequences of numbers is expected to provide significant memory savings. Generally, interval lists use two 32-bit integers to store a single interval. It might be possible to reduce this memory footprint by looking at other ways to store contiguous sequences of numbers. In this section we introduce bit vector compression for storing reachability information. The principal contribution of this paper is the application of such compression to storing transitive closures; the invention of a novel compression scheme that is particularly well-suited to that application is a secondary contribution.

As we have just seen, constructing a transitive closure by using Nuutila's algorithm basically boils down to merging reachability information of strongly connected components a large number of times. When using bit vectors (either compressed or regular), merging reachability information is done by computing a logical OR of the bit vectors storing reachability information of adjacent strongly connected components. For example, the reachability bit vector of component C_5 in Figure 1 is constructed by computing the logical OR of the bit vectors containing reachability information of components C_3 and C_4. Based on this observation, it is important for a compression scheme to provide good performance of a logical OR operation in order to facilitate efficient construction of the transitive closure data structure.

3.2 WAH

3.2.1 General introduction

In [22], Wu *et al.* present a comparison study of a range of bit vector compression schemes. Based on an experimental and theoretical analysis they show that the *Word Aligned Hybrid* compression scheme (WAH) provides a significant compression whilst maintaining the ability to do fast bitwise operations like AND, OR and XOR. WAH has found myriad applications, an overview of these can be found in [21].

The WAH compression scheme processes an input bit vector *run-length* in blocks of $b = w - 1$ bits, w denoting the word size. Wu *et al.* assume a word consists of 32 bits, resulting in a block size of 31 bits. We will adapt these assumptions for describing WAH, unless stated otherwise.

When compressing a bit vector using WAH, each input block can be stored in one of two ways:

1. Using a *fill word* – indicated by a preceding 1-bit – if the block entirely consists of b consecutive 0-bits or 1-bits. A fill word simply stores (1) the type of fill (1-bits or 0-bits) and (2) the number of contiguous bits, as a multiple of b.

2. Using a *literal word* – indicated by a preceding 0-bit – if the block consists of a mixed sequence of 0-bits and 1-bits.

Figure 2 illustrates how an input bit vector is compressed by the WAH compression scheme. The underlined bits store the type of the word (0 = literal, 1 = fill) and the type of the fill word: a 0-fill for storing a sequence of 0-bits, a 1-fill for storing a sequence of 1-bits.

3.2.2 Merging reachability information

As has been indicated earlier, the reachability information stored for a component C_k is constructed by computing the bitwise logical OR of the reachability bit vectors of its directly adjacent strongly connected components (see Figure 1). It is possible to design a very efficient OR operation for WAH compressed bit vectors, which turns out to work as least as efficient as a bitwise logical OR on equivalent uncompressed bit vectors.

In the worst case, the bit vector is not suitable for compression (*i.e.*, there are no parts suitable for being stored using a 0-fill or 1-fill). In that case, WAH will introduce an overhead linear in the size of the bit vector: one extra 0-bit

Uncompressed input bit vector (in groups of 31 bits):

```
0000000000000000000000000000000 0000000000000000000000000000000
0000000000100101000010000100111 1111111111111111111111111111111
1111111111111111111111111111111 1111111111111111111111111111111
```

WAH compressed bit vector (using 3 words of 32 bits):

$$\underbrace{\underline{\texttt{10}}\texttt{000000000000000000000000000010}}_{\text{0-fill: 2 blocks of contiguous 0-bits}}\ \underbrace{\underline{\texttt{0}}\texttt{0000000000100101000010000100111}}_{\text{literal word: a block of mixed bits}}$$

$$\underbrace{\underline{\texttt{11}}\texttt{000000000000000000000000000011}}_{\text{0-fill: 3 blocks of contiguous 1-bits}}$$

Figure 2: WAH compression example

(to indicate the type of word: literal) for each block of 31 bits. Computing the bitwise logical OR of two of those bit vectors is a fairly trivial operation. Although the worst-case complexity of this operation is equal to the complexity of a bitwise OR on two uncompressed bit vectors, its performance becomes more attractive when a bit vector contains multiple fill words. For example, a very large 1-fill – which is expressed using a single word of 32 bits – will effectively reduce the number of OR operations.

3.2.3 Limitations of WAH

The WAH compression scheme allocates $b - 1 = w - 2$ bits of a fill word to store the length of a fill, which itself is expressed in multiples of $b = w - 1$. For a word size of 32 bits, 30 bits are available solely for storing the length of a fill, yielding a maximum fill length of $2^{30} - 1 = 1\,073\,741\,823$ blocks ($= 33\,285\,996\,513$ consecutive identical input bits). When using a word size of 64 bits, the maximum fill length which can be expressed increases to $(2^{62} - 1) \times 63 \approx 2.91 \times 10^{20}$ bits. It is evident that fills of both sizes are never encountered when computing and storing a transitive closure, even if the input graph contains millions of vertices. Since these 30 bits will never be fully employed, a lot of memory is wasted on storing zeroes.

Another significant disadvantage of WAH is its granularity: the compression scheme is quite coarse grained. Consider the following input bit vector:

$$\underbrace{\texttt{0111111111111111111111111111111}}_{\text{0-bit followed by 30 1-bits}}\ \underbrace{\texttt{1111111111111111111111111111110}}_{\text{30 1-bits followed by a 0-bit}}$$

Although the example input bit vector contains 60 contiguous 1-bits, WAH will not be able to compress the bits since it reads the input block by block. A block size of $w - 1 = 31$ bits is quite large, resulting in a rather coarse grained compression scheme.

Reducing the block size (*i.e.* no longer fix it to $b = 31$ bits) will solve both problems:

- less bits will be allocated for storing fill lengths;
- less contiguous bits are required in order to be able to create a fill word.

A naive reduction in block size could, however, prevent us from representing long intervals. The next section shows how we circumvent that problem.

$$\underbrace{\texttt{1010}}_{\text{hdr}}\ \underbrace{\texttt{100000000001011}}_{\text{1-fill: 11 blocks}}\ \underbrace{\texttt{111010000100101}}_{\text{literal: 15 bits}}\ \underbrace{\texttt{000000000010010}}_{\text{0-fill: 18 blocks}}\ \underbrace{\texttt{000000010011100}}_{\text{literal: 15 bits}}$$

$$\underbrace{\texttt{0101}}_{\text{hdr}}\ \underbrace{\texttt{000111011001010}}_{\text{literal: 15 bits}}\ \underbrace{\texttt{000000000001010}}_{\text{0-fill: 10 blocks}}\ \underbrace{\texttt{010101110100011}}_{\text{literal: 15 bits}}\ \underbrace{\texttt{100000000110011}}_{\text{1-fill 51 blocks}}$$

Figure 3: Example PWAH-4 compressed bit vector (two words, four partitions each)

3.3 PWAH

3.3.1 Introduction

We introduce PWAH: Partitioned Word Aligned Hybrid compression. This new compression scheme very much resembles WAH: it distinguishes between compressing a sequence of bits as a fill, or storing the bits as a literal. However, a word (which is assumed to consist of 64 bits) is divided into P *partitions*. Furthermore, a word contains a *header* of P bits, which specifies the type (literal or fill) of each of the partitions.

Figure 3 shows an example PWAH-4 compressed bit vector, consisting of two (64-bit) words and eight partitions.

3.3.2 Partitions

PWAH supports two, four and eight partitions of size $\lfloor \frac{64-P}{P} \rfloor$ bits. Hence, it is actually a collection of compression schemes, each having its own properties. The most important properties are listed in Table 1.

It is not hard to see that any other number of partitions would result in bits not being used. For example, PWAH with 13 partitions would yield a partition size of $\lfloor \frac{64-13}{13} \rfloor = 3$ bits, wasting 12 bits per word. Note that $P = 16$ or $P = 32$ denote valid numbers of partitions in the sense that no bits are left unused. However, both schemes introduce a quite substantial overhead (25% and 50% respectively), used to store the word header. More importantly, a partition consists of only $\frac{64-16}{16} = 3$ bits in PWAH-16. When storing a fill partition, the PWAH-16 scheme leaves only 2 bits for expressing the fill length, which is generally insufficient. Therefore, the PWAH-16 and PWAH-32 schemes are not considered.

As listed in Table 1, the block size of the PWAH schemes is equal to the respective partition sizes. A smaller block size allows us to compress a bit vector in a more fine grained way. However, decreasing the block size has a twofold effect on the maximum fill length:

1. less bits are available for expressing the fill length;
2. a smaller block size implies less bits per block and hence less bits per fill of n blocks. *E.g.*, a fill of size n blocks stores $n \times 15$ contiguous bits in PWAH-4, but only $n \times 7$ bits in PWAH-8.

Therefore, the actual number of bits which can be expressed in PWAH decreases rapidly when the block size decreases. As can be seen in Table 1, the maximum number of contiguous input bits which can be expressed in a PWAH-8 fill is only $(2^6 - 1) \times 7 = 441$ bits. Although the small block size results in a more fine grained compression scheme, it is lacking support for large fills.

3.3.3 Extended fills

To be able to use fills representing more than 441 contiguous bits whilst maintaining a fine grained compression

	PWAH-2	**PWAH-4**	**PWAH-8**
Partitions	2	4	8
Part. size	31 bits	15 bits	7 bits
Block size	31 bits	15 bits	7 bits
Max. fill	3.33×10^{10} bits	245 745 bits	441 bits

Table 1: Important properties of PWAH schemes

scheme, we introduce *extended fills*: fills that span multiple partitions. Instead of summing the length of two (or more) consecutive fill partitions, the bits from the partitions are *concatenated* before being interpreted.

For example, consider five example PWAH-8 partitions:

$$\underbrace{\texttt{01110}\ldots}_{\text{hdr}}\ \underbrace{\texttt{0010111}}_{\text{literal}}\ \underbrace{\underline{\texttt{1}}\texttt{001011}}_{\text{fill (1/3)}}\ \underbrace{\underline{\texttt{1}}\texttt{010001}}_{\text{fill (2/3)}}\ \underbrace{\underline{\texttt{1}}\texttt{001010}}_{\text{fill (3/3)}}\ \underbrace{\texttt{0101001}}_{\text{literal}}\ \underbrace{\ldots\ldots}$$

The above example contains 3 1-fill partitions, embraced by two literals. In stead of interpreting these 1-fills individually as $\texttt{0b001011} + \texttt{0b010001} + \texttt{0b001010} = 11 + 17 + 10 = 38$ blocks of contiguous 1-bits, the bits are concatenated: $\texttt{001011} \circ \texttt{010001} \circ \texttt{001010} = \texttt{0b001011010001001010} = 46\,154$ blocks (using $\circ$ as the concatenation operator).

Note that there is no need to explicitly identify an extended fill: every sequence of identical fills can be considered to be an extended fill and can be interpreted as such. When reading a PWAH bit vector, it takes only a constant number of extra operations to determine whether a fill is spanning multiple partitions or not. Therefore, the time complexity of reading a PWAH bit vector is not affected by this additional feature.

Using extended fills in PWAH-8, all eight partitions (each providing six bits for expressing the length of the fill) can be employed to denote a single fill of length $2^{8\times 6} - 1 \approx 2.815 \times 10^{14}$ blocks $\approx 1.970 \times 10^{15}$ contiguous bits. Applying extended fills to PWAH-8 yields a very fine-grained compression scheme, without posing a limit on the length of the fills.

4. THEORETICAL ANALYSIS

4.1 General

Nuutila presents a very detailed asymptotic runtime analysis of his algorithm in [11]. It is clear that the performance of the algorithm strongly depends on the data structure used to store reachability information. In terms of construction time, there is no difference in the asymptotic complexity of interval lists and PWAH compressed bit vectors. In this section we will show that there actually is a clear asymptotic difference in query time performance and we will prove that PWAH-8 will virtually always outperform interval lists in terms of memory usage.

4.2 PWAH

4.2.1 *Transitive closure construction*

Constructing a PWAH bit vector is done by merging two other bit vectors. This operation requires a single full scan over both bit vectors, processing the vectors block by block. The number of blocks is linear in the number of uncompressed bits and therefore, the merge operation is linear in the size of the input vectors.

$$\underbrace{\texttt{0100}}_{\text{hdr}}\ \underbrace{\texttt{001001001001011}}_{\text{literal: 15 bits}}\ \underbrace{\texttt{100000001010001}}_{\text{1-fill: 81 blocks}}\ \underbrace{\texttt{100010100110010}}_{\text{literal: 15 bits}}\ \underbrace{\texttt{001001010101100}}_{\text{literal: 15 bits}}$$

$$\underbrace{\texttt{0110}}_{\text{hdr}}\ \underbrace{\texttt{001010100110010}}_{\text{literal: 15 bits}}\ \underbrace{\texttt{000000001001011}}_{\text{part 1 of 0-fill}}\ \underbrace{\texttt{001000100000101}}_{\text{part 2 of 0-fill}}\ \underbrace{\texttt{001001010100011}}_{\text{literal: 15 bits}}$$

index pos.	1	2	3	$\cdots$	18,065
index bit	1,024	2,048	3,072	$\cdots$	18,498,560
global block	$\lfloor \frac{1024}{15} \rfloor = 68$	136	204	$\cdots$	1,233,237
word	0	1	1	$\cdots$	1
partition	1	2	2	$\cdots$	2
part. block	67	51	119	$\cdots$	1,233,152

Figure 4: Indexing two PWAH-4 words compressing 18,498,645 bits, using an index chunk size of 1024 bits

The condensation graph $G_c = (V_c, E_c)$ of an input graph $G = (V, E)$ on which the merge operations are performed is acyclic by definition and can therefore contain at most $\frac{n_c \times (n_c - 1)}{2}$ edges ($n_c = |V_c|$). The asymptotic worst-case construction time in terms of the number of vertices is therefore $\mathcal{O}(n_c^3)$. Note that we have temporarily ignored the possible existence of (a total of up to n_c) self-loops on strongly connected components, but these are insignificant to the $\mathcal{O}(n_c^3)$ asymptotic bound.

Alternatively, the bound can be expressed in terms of the number of vertices $n = |V|$, the number of edges $m = |E|$ and the number of vertices and edges in the condensation graph n_c and $m_c = |E_c|$ as $\mathcal{O}(n + m + n_c m_c)$.

4.2.2 *Query time – regular PWAH*

In contrast to the constant time required to execute a random query on a regular bit vector, PWAH compressed bit vectors can answer random queries in time linear to the size of the vector. This is caused by the fact that bits are no longer stored at a specific offset in memory: the memory location of a bit b_i strongly depends on the compression of the blocks storing all bits up to and including bit b_i. Therefore, a random query might require a full scan over the bit vector.

A sorted series of any number of queries can be answered in linear time, by performing one full scan over the PWAH compressed bit vector.

4.2.3 *Query time – indexed PWAH*

A well-known technique for improving query time is indexing. 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 PWAH bit vector can be considered constant. Introducing indices will increase memory usage of the compressed bit vector, but only linearly. Hence, the asymptotic memory usage does not change: $\mathcal{O}(l)$.

An example of an indexed PWAH-4 bit vector is shown in Figure 4. Suppose the value of bit b_j (j denoting the index of the bit) is queried. When using PWAH without indices, this query would require a scan starting at the first bit. Using the index it is now possible to start scanning at the closest smaller indexed bit b_i ($i \leq j$) by looking at index position $\lfloor \frac{j}{k} \rfloor$ and commence seeking from the partition in which b_i resides, until b_j is encountered. This would require processing at most k bits, stored in at most $\frac{k}{15}$ PWAH-4 partitions (15 being the block size in PWAH-4).

For example, a lookup of bit 3100 would require a scan starting at the bit at index entry $\lfloor \frac{3100}{1024} \rfloor = 3$. The bit at index entry 3 has bit index 3072 and resides in word 1, partition 2, block offset 119 of the PWAH compressed vector. Bit 3100 resides within block $\lfloor \frac{3100}{15} \rfloor = 206$, which is only $206 - 204 = 2$ blocks away from bit 3072.

The performance of PWAH indexing will be subject of further evaluation in Section 6.

4.2.4 *Memory usage*

In the worst case, the input to the PWAH compression scheme consists of a sequence of scattered bits which is not suitable for compression. In that case, PWAH induces an overhead linear in the number of input bits k. Therefore, the worst case memory usage is $\mathcal{O}(k)$. Although the expected memory usage is significantly smaller, it is very hard to prove a meaningful bound on that.

4.3 Interval lists

4.3.1 *Transitive closure construction*

Merging two interval lists can be done in linear time by scanning each input list only once and merging overlapping (or directly adjacent) intervals in the process.

4.3.2 *Query time*

When processing the strongly connected components in reverse topological order, the resulting interval lists will automatically be sorted. Therefore, a simple binary search can be employed to answer queries on the interval list, yielding a worst-case query time complexity of $\mathcal{O}(\log_2 k)$ (with k denoting the number of intervals, which is linear in the number of indices). A sorted series of any number of queries on a single interval list can be answered in $\mathcal{O}(k)$ time by a full scan over the interval list.

4.3.3 *Memory usage*

In the worst case, an interval lists consists of intervals of size 1. It is not hard to see that the number of intervals required to store a subset I' of a set of integers $I = \{1, 2, \ldots, k\}$ is at most $\lceil \frac{k}{2} \rceil$. The memory usage of interval lists is therefore linear in the number of input integers k.

4.4 Interval lists vs. PWAH: memory usage

4.4.1 *Differences and similarities*

The compression technique of interval lists is very similar to the PWAH compression scheme: both approaches provide a way to compress sequences of contiguous identical bits. However, there are two significant differences:

- Interval lists store the begin and end of a sequence of contiguous bits explicitly by storing the offsets as integers, PWAH does not store an explicit left and right bound and therefore needs to store non-existing edges (by using 0-bits) as well;
- PWAH stores the length of a sequence of contiguous bits as a multiple of the block size (hence the term 'aligned'), interval lists can store a sequence at any offset of any length.

Given these differences we analyse the memory usage of PWAH compared to interval lists.

$$I = \{(5, 49), (94, 151)\}$$

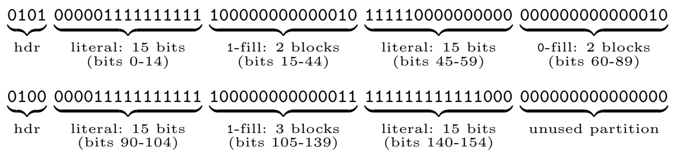

0101
hdr
000001111111111
literal: 15 bits (bits 0-14)
100000000000010
1-fill: 2 blocks (bits 15-44)
111110000000000
literal: 15 bits (bits 45-59)
000000000000010
0-fill: 2 blocks (bits 60-89)
0100
hdr
000011111111111
literal: 15 bits (bits 90-104)
100000000000011
1-fill: 3 blocks (bits 105-139)
111111111111000
literal: 15 bits (bits 140-154)
000000000000000
unused partition

Figure 5a: Example interval list and equivalent PWAH-4 bit vector

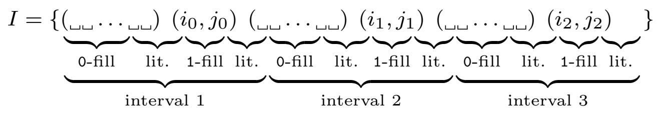


Figure 5b: Generalised way of expressing intervals in PWAH

4.4.2 *Expressing an interval list using PWAH*

Consider the example interval list and equivalent PWAH-4 compressed bit vector shown in Figure 5a.

Although an interval and a PWAH fill partition show a remarkable resemblance, it is clear that in most cases a single interval can not be represented by a single fill partition. In the example shown in Figure 5a, a total of seven PWAH-4 partitions were required to represent the information captured in the interval list.

It is possible to generalise this example, as depicted in Figure 5b. The actual intervals in the interval list I are (i_0, j_0), (i_1, j_1) and (i_2, j_2). The gaps between the three intervals (which can be considered to be the 0-bits in the equivalent bit vector) are denoted by $(\sqcup\sqcup \cdots \sqcup\sqcup)$.

From the figure, it follows that – worst-case – a single interval (i_k, j_k) in an interval list I requires four PWAH parts:

1. A preceding 0-fill z_k;
2. A preceding literal partition m_k (consisting of mixed bits), to glue z_k and o_k together;
3. A 1-fill o_k to represent the core of the interval;
4. A succeeding literal partition m'_k, to glue o_k and z_{k+1} (preceding the next interval) together.

Note that this describes the worst-case scenario. Examples of less bad cases are (but are not limited to):

- Two intervals (i_k, j_k) and (i_{k+1}, j_{k+1}) lie very close to eachother: z_{k+1} can be omitted;
- An interval (i_k, j_k) is very small: o_k and m'_k can be omitted, the interval is captured within m_k.

We now observe that – worst-case – an interval list $I = [(i_0, j_0), \ldots, (i_{n-1}, j_{n-1})]$ consisting of n intervals needs to be represented by n repetitions of [0-fill, literal, 1-fill, literal]. This sequence requires at least four PWAH partitions, but possibly more in case one or both of the 1-fill and 0-fill span multiple partitions (by using extended fills). As can be seen in Table 1, PWAH-8 can store regular (non-extended) fills representing up to 441 contiguous identical bits. If either the gap (0-fill) z_k preceding an interval k or the interval itself (represented by the 1-fill o_k) exceeds that number of bits, PWAH-8 will need to employ extended fills – hence using more than four partitions – to store the interval o_k and its preceding gap z_k.

Four PWAH partitions require 128 bits (PWAH-2), 64 bits (PWAH-4) or 32 bits (PWAH-8). A straightforward implementation of an interval list would use two 32-bit integers to store a single interval, a total of 64 bits per interval. Therefore, in terms of memory usage, only PWAH-8 is an interesting competitor.

The following two theorems (and proofs) provide some grips on reasoning about interval lists in terms of PWAH compressed bit vectors:

Theorem 1. *Using a PWAH scheme with block size b, the number of partitions q required to store a sequence of n contiguous identical bits (n a multiple of b) equals:*

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

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. $\square$

Theorem 2. *Using a PWAH scheme with block size b, the total number of partitions p required to store an interval (i_k, j_k) of length $n_1 = j_k - i_k + 1$ and its preceding gap of length $n_0 = i_k - j_{k-1} - 1$ equals:*

$$\begin{aligned} p(n_0, n_1, b) &\leq 2 + q(n_0, b) + q(n_1, b) \\ &= 2 + \left\lceil \frac{\lceil \log_2(\frac{n_0}{b}+1)\rceil}{b-1} \right\rceil + \left\lceil \frac{\lceil \log_2(\frac{n_1}{b}+1)\rceil}{b-1} \right\rceil \end{aligned}$$

Proof. At most one partition is used to concatenate the 0-fill representing the gap and the 1-fill representing the interval (i_k, j_k), another partitions is used to concatenate the 1-fill to the 0-fill representing the next gap (between (i_k, j_k) and (i_{k+1}, j_{k+1})). The rest of the theorem follows from the proof of Theorem 1. $\square$

It follows from Theorems 1 and 2 that in order for PWAH-8 to use less memory than an interval list I, it is required that $p(n_0, n_1, 7) < 8$ holds for intervals $(i_k, j_k) \in I$. The plot shown in Figure 6a depicts the relation between the amount of memory used by an interval list to store a single interval (in terms of number of PWAH-8 partitions: 8 partitions = 64 bits) and the amount of memory an equivalent PWAH-8 compressed bit vector would use. Note that the p is considered a continuous function, rather than being discontinuous due to the *ceil* operands.

Figure 6b illustrates the (discontinuous) dependency between the size of the gap and the interval. As can be seen in the figures, PWAH-8 outperforms interval lists, unless both the gap and the interval are extremely large. As will be shown in Section 6, this situation has not occurred in our experiments.

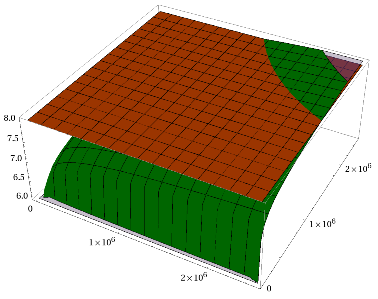


Figure 6a: Memory required to store an interval using interval lists (red plane) versus the equivalent PWAH-8 representation (green surface)

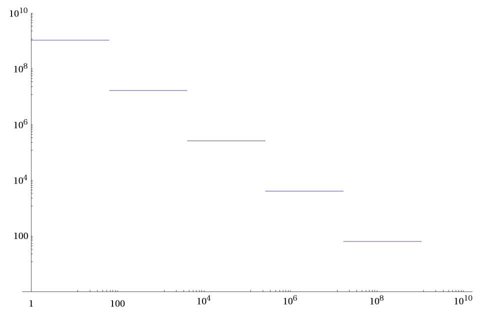


Figure 6b: Dependency of maximum 1-fill (interval) and 0-fill (gap) sizes in order to save memory

5. RELATED WORK

Data structures for the computation of transitive closure have a long and rich history. Standard approaches based on matrix multiplication exhibit very good worst-case complexity, but they are not practical due to memory requirements. An overview of techniques to make the Floyd-Warshall algorithm more practical [20, 16, 5] can be found in [2].

As we observed in Section 2, a key step towards a practical algorithm is to conduct the computation on the graph of strongly connected components. The first publication to make that observation was by Purdom [13].

Since then, a substantial number of improvements to Purdom's basic scheme were presented, in particular by Eve and Kurki-Suonio [4], Schmitz [17] and Ioannidis *et al.* [7]. This line of work culminated in the discovery, by Nuutila and Soisalon-Soinen, of a single pass algorithm that was shown to be superior to all available alternatives in experimental evaluation around 1994 [12, 10, 11]. The present paper directly builds on this work. In [6] it is shown how Nuutila's algorithm can be refined to provide better performance in a layered memory architecture, in particular by merging several stacks into one. We have not used that more involved version of the algorithm, instead electing to use the simpler version from Nuutila's dissertation. It is clear, however, that our use of PWAH carries over to the variation in [6].

The first paper that considered the use of compression in representing transitive closures was [1]. That paper considers different ways of labelling vertices to achieve better compression with interval lists. As we have remarked, it is a by-product of Nuutila's algorithm that reasonably good compression is achieved in practice. Indeed, in most of the above works, in particular those of Nuutila, the chosen data structure for representing adjacency information is interval lists. It is the exploitation of that special characteristic of transitive closure to achieve good compression that sets the present paper apart from other applications of WAH and its variants (such as full text search [18]).

A completely different approach to compactly representing reachability was proposed by Cohen *et al.* [3]. The key concept is that of a *2-hop cover*: a collection of paths S such that for any two vertices u and v, if there is a path from u to v there exists a concatetation of two paths in S from u to v.

New applications in XML processing, biological databases, social network analysis, semantic web and so on spurred on a renewed interest in transitive closure computation in recent years, resulting in multiple papers at SIGMOD and elsewhere. As these are the most recent significant advances in the area, it behooves us to review them in some detail.

The Path-Tree [9] approach employs a sequence of nontrivial steps to construct a 'path-tree cover': a tree-shaped spanning subgraph of the input graph. Using this cover it is possible to attach two labels to every vertex, allowing efficient query processing: constant time for a limited subset of pairs of vertices, but $\mathcal{O}(\log_2 k)$ in general (k denoting the number of paths in the path decomposition). Furthermore, the authors introduce the concept of an *optimal* path decomposition by showing that the problem is actually similar to the minimal-cost flow problem.

The Path-Tree scheme was further generalised by Zhu *et al.* [23]: they present a general framework partitioning the input graph, and then indexing each subgraph separately. By chosing a particular partitioning heuristic, they achieve somewhat better memory efficiency than [9] on large graphs that are not tree-like. The main contribution of [23] is however not in that better performance, but in providing a framework that allows different indexing techniques to be used. Indeed it would be interesting to explore the combination with PWAH.

At SIGMOD 2009 (one year after introducing Path-Tree), Jin *et al.* introduced 3-HOP [8]. The algorithm uses chain decomposition to identify chains which can be seen as highways of a graph. When answering a reachability query to determine whether a vertex v can reach w, the algorithm tries to reach an appropriate highway chain (hop 1), travel across the chain and take a suitable 'exit' (hop 2) and finally reach w (hop 3). This is an improvement of the 2-hop approach of Cohen *et al.* [3] discussed above. The authors argue that 3-HOP is designed for dense graphs, in which case other approaches like Path-Tree are not performing well.

We shall carefully evaluate and contrast PWAH with the work of Jin *et al.* in Section 6. In particular we shall use the benchmarks from [9, 8] in our own experimental evaluation.

6. EXPERIMENTAL EVALUATION

6.1 Introduction

6.1.1 Structure of experiments

In this section, we evaluate the performance of PWAH (with focus on PWAH-8) in two distinct ways. Firstly, randomly generated graphs are used to investigate the behaviour of PWAH with respect to varying graph topologies. Secondly, the performance of our PWAH implementation is

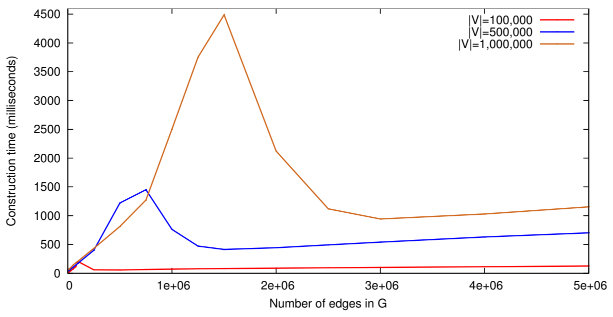


Figure 7: Construction time of a transitive closure using PWAH-8, depending on the number of vertices and edges in a randomly generated graph

compared to that of our interval lists implementation and the Path-Tree[9] and 3-HOP implementations by Jin *et al.*, using real-world graphs. Generally, we look at construction time, query time and memory usage.

6.1.2 Hardware and software

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

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.

6.2 PWAH behaviour depending on input

6.2.1 Generating random graphs

In order to be able to evaluate the behaviour of PWAH transitive closure compression with respect to graphs of different sizes and with different topologies, a large data set of random graphs has been generated. The number of vertices in the input graphs ranges from 100 000 to 2 000 000, with a number of edges ranging from 10 000 to 10 000 to 5 000 000. Some of the figures presented in this section are based on a smaller subset of the generated graphs, to illustrate specific interesting (sometimes unexpected) effects.

The process of generating a random graph with n vertices and m edges consists of two steps:

1. Create a graph $G = (V, E)$ with $|V| = n$ vertices and an empty edge set: $E = \varnothing$;

2. Randomly choose two vertices v, w. If $(v, w) \notin E$, add an edge (v, w) to E and repeat this step until $|E| = m$.

Note that for a sufficiently small number of edges, the number of strongly connected components will equal the number of vertices: $|V| = |C|$ (*i.e.*, every strongly connected component consists of only one vertex). Gradually increasing number of edges will eventually yield a decreasing number of strongly connected components.

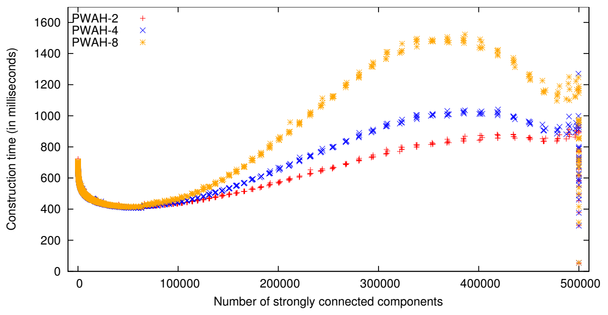


Figure 8: Construction time w.r.t. number of strongly connected components (number of vertices is fixed, increasing number of edges: $|V| = 500\,000$, $10\,000 \leq |E| \leq 5\,000\,000$)

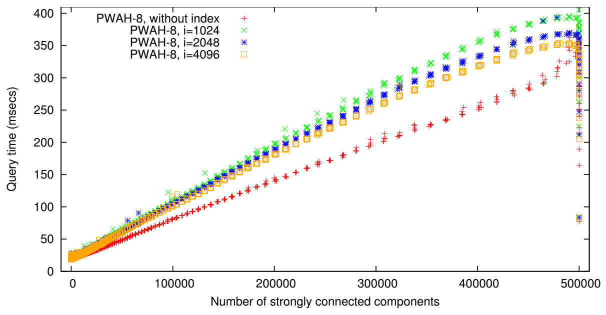


Figure 9: Query time of a transitive closure stored using PWAH-8 and three different indexing strategies

6.2.2 *Construction time*

Figure 7 shows the construction time of a PWAH-8 compressed transitive closure data structure for a number of different vertex and edge counts. The figure shows that these counts have a very strong effect on the construction time, which is not surprising.

Recall that Nuutila's algorithm will perform a merge operation on roughly each edge in the condensation graph. An increasing number of edges will initially *not* result in a smaller number of strongly connected components, but will of course increase connectivity between strongly connected components and therefore increase the processing time. After having reached a critical edge density, randomly adding more edges will yield larger (and therefore less) strongly connected components. As the number of edges increases, the number of components drops drastically and the number of merge operations will decrease, resulting in a decreasing construction time. At the local minimum, the number of components is still decreasing, but only very slowly. The time gained by having to perform less merge operations does no longer dominate the costs of graph traversal and the plot line shows a (linearly, because of the DFS) increasing construction time.

Given the structure of Nuutila's algorithm, it is not surprising that the construction time appears to depend on the number of strongly connected components. Figure 8 shows the construction time of the transitive closure data structure with respect to the number of strongly connected components, using a range of PWAH compression schemes to store reachability information.

6.2.3 *Query time*

As has been shown in Section 4, the lookup time of a random bit in a PWAH-8 compressed bit vector is linear in the size of the vector. To be able to verify this asymptotic performance, the total running time of 1 000 000 random queries has been measured. The outcome of this experiment confirms the theoretical results from Section 4, as can be seen in Figure 9 (the red data points depict PWAH-8 without indexing: $+$).

Although introducing indexing was expected to yield a constant lookup time, this theory can not be confirmed using the experiments on random graphs. Though initially unexpected, this behaviour can be explained by looking at the bit vectors a little bit more closely. Most PWAH bit vectors only contain a very limited amount of partitions (mostly tens, sometimes a few hundred), whilst representing a very large number of bits. The accompanying indices are therefore relatively large, resulting in bad caching behaviour for random index lookups. Since the actual size of the PWAH bit vectors is rather small, a linear scan operation can be performed within reasonable time, actually taking full advantage of caching.

6.3 Performance comparison

6.3.1 *Input graphs*

A data set consisting of several real-world graphs has been used to compare PWAH to interval lists, Path-Tree and 3-HOP. The details of these input graphs are listed in Table 2.

The graphs can be divided into roughly three categories:

- Graphs used for source code analysis, kindly provided by Semmle Ltd.[1], based on four open-source projects: ADempiere[2], ImageMagick[3], Firefox and Samba[4];
- Graphs describing the structure of Wikipedia pages and categories, provided by Semmle Ltd. and based on the Simple English Wikipedia of July 1st 2009;
- Graphs used in two papers by Jin *et al.* [9, 8].

Table 2 provides information about five different topological properties:

$$\begin{aligned} |V| &= \text{Number of vertices;} \\ |E| &= \text{Number of edges;} \\ |V_C| = |C| &= \text{Number of strongly connected components (vertices in the condensed graph);} \\ |E_C^+| &= \text{Number of edges between strongly connected components in the transitive closure;} \\ |E^+| &= \text{Number of edges in the transitive closure;} \end{aligned}$$

[1]Semmle Ltd: `http://www.semmle.co.uk`
[2]ADempiere: a commons-based peer-production of open-source ERP applications (version 3.5.1a): `http://www.adempiere.org`
[3]ImageMagick: a software suite to create, edit, and compose bitmap images (version 6.5.7-8): `http://www.imagemagick.org`
[4]Samba: a Windows interoperability suite of programs for Linux and Unix (version 3.2.23): `http://www.samba.org`

Unfortunately, both Path-Tree approaches described in [9] turned out to be incapable of processing the largest graphs in the data set: in some cases the implementation exited unexpectedly with a *segmentation fault*, in other cases Path-Tree did not succeed in executing five runs within three minutes. Path-Tree bars for these graphs are missing in Figure 10.

The 3-HOP[8] approach did time out on virtually *all* input graphs from our data set (including the graphs which were used by Jin *et al.* when presenting Path-Tree). Therefore, 3-HOP is not included in the performance comparison shown in Figure 10. Later in this section, 3-HOP will be compared to PWAH-8 and interval lists using the graphs from [8].

6.3.2 *Construction time*

Looking at the bar charts in Figure 10, it is clear that Nuutila's algorithm with PWAH-8 and interval lists are outperforming both Path-Tree approaches by far. Note that the Y-axis has been scaled logarithmically, showing a difference in performance of about two orders of magnitude in some cases.

PWAH-8 and interval lists show quite similar performance, except for the largest (Firefox) graphs in which PWAH-8 performs significanly better in terms of construction time.

6.3.3 *Query time*

To test the query performance of the different approaches, a total of 1 000 000 random queries were executed. As depicted in Figure 10, the differences in performance are not very significant in most cases. In general, both interval lists and PWAH-8 slightly outperform Path-Tree, with interval lists showing the most impressive results.

This result is rather remarkable, since Jin *et al.* claim an $\mathcal{O}(1)$ lookup time (which turns out to be $\mathcal{O}(\log_2 k)$ as mentioned in Section 5). Recall from Section 4 that interval lists provide an $\mathcal{O}(\log_2 n)$ query time, and PWAH-8 takes $\mathcal{O}(n)$ time. Based on this theoretical analysis, one would expect Path-Tree to significantly outperform both interval lists and PWAH-8, which is clearly not the case.

6.3.4 *Memory usage*

The experiment results on memory usage confirm one of the most important theoretical result introduced from Section 4: PWAH-8 will (virtually) always outperform interval lists in terms of memory usage. Figure 10 suggests that PWAH-8 yields a significantly higher compression than Path-Tree on the graphs contained in the data set.

6.3.5 *3-HOP*

As stated before, the 3-HOP implementation by Jin *et al.* [8] failed to process almost all input graphs from our data set. Therefore, 3-HOP is compared to PWAH-8 and interval lists solely using the graphs with which it was presented at SIGMOD 2009. The graphs used in [8] to evaluate the performance of 3-HOP are relatively small: little over 10 000 vertices and 67 000 edges (see Table 2). It is worth noticing that these graphs are acyclic, hence the number of strongly connected components equals the number of vertices.

It takes PWAH-8 and interval lists at most 100 milliseconds per graph to construct the transitive closure, whereas 3-HOP needs at least 11 seconds to little over 17 minutes

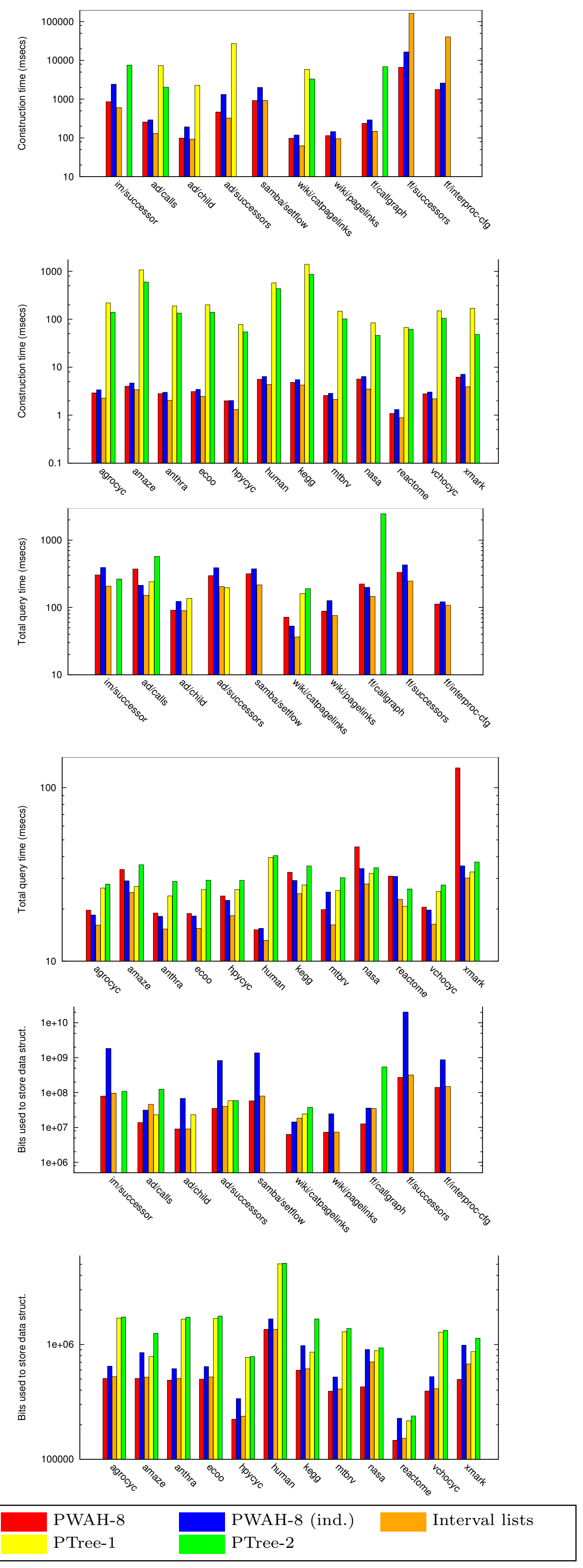


Figure 10: Performance comparison (log scale)

		$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V_C\rvert = \lvert C\rvert$	$\lvert E_C^+\rvert$	$\lvert E^+\rvert$
Static source code analysis						
ADempiere	Calls	52 290	198 041	52 225	8 726 099	8 972 548
	Child	181 091	180 373	181 091	568 468	568 468
	Successors	423 358	448 909	384 094	4 014 609	7 323 367
ImageMagick	Successors	1 095 062	1 145 304	542 235	120 800 042	2 309 714 078
Samba	Setflow	568 656	884 839	510 720	3 134 420 405	10 968 067 063
Firefox	Callgraph	70 241	276 960	64 325	108 542 373	290 755 498
	Successors	3 525 673	4 138 475	1 858 504	252 357 804 602	3 789 586 691 341
	Interproc. CFG	3 532 298	4 716 476	353 748	23 167 640 997	11 013 708 014 351
Other						
Wikipedia	Cat. pagelinks	75 946	181 084	75 936	2 209 952	2 262 118
	Pagelinks	137 830	2 949 220	47 242	104 513 953	12 479 685 213
Jin *et al.*						
Path-Tree[9]	AgroCyc	13 969	17 694	12 684	170 591	2 731 596
	Amaze	11 877	28 700	3710	2 371 476	93 685 094
	Anthra	13 736	17 307	12 499	148 559	2 440 124
	Ecoo	13 800	17 308	12 620	173 254	2 398 250
	HpyCyc	5565	8474	4771	77 131	1 113 356
	Human	40 051	43 879	38 811	348 112	2 804 552
	Kegg	14 271	35 170	3617	2 637 440	147 922 066
	Mtbrv	10 697	13 922	9602	139 275	2 038 237
	Nasa	5704	7939	5605	167 243	186 900
	Reactome	3678	14 447	901	32 607	6 666 940
	VchoCyc	10 694	14 207	9491	136 674	2 343 636
	Xmark	6483	7954	6080	536 389	2 307 574
3-HOP[8]	ArXiv	6000	66 707	6000	5 566 205	5 566 205
	CiteSeer	10 720	44 258	10 720	421 995	421 995
	GO	6793	13 361	6793	104 178	104 178
	PubMed	9000	40 028	9000	523 037	523 037
	YAGO	6642	42 392	6642	66 439	66 439

Table 2: Characteristics of non-artificial graphs used for experiments

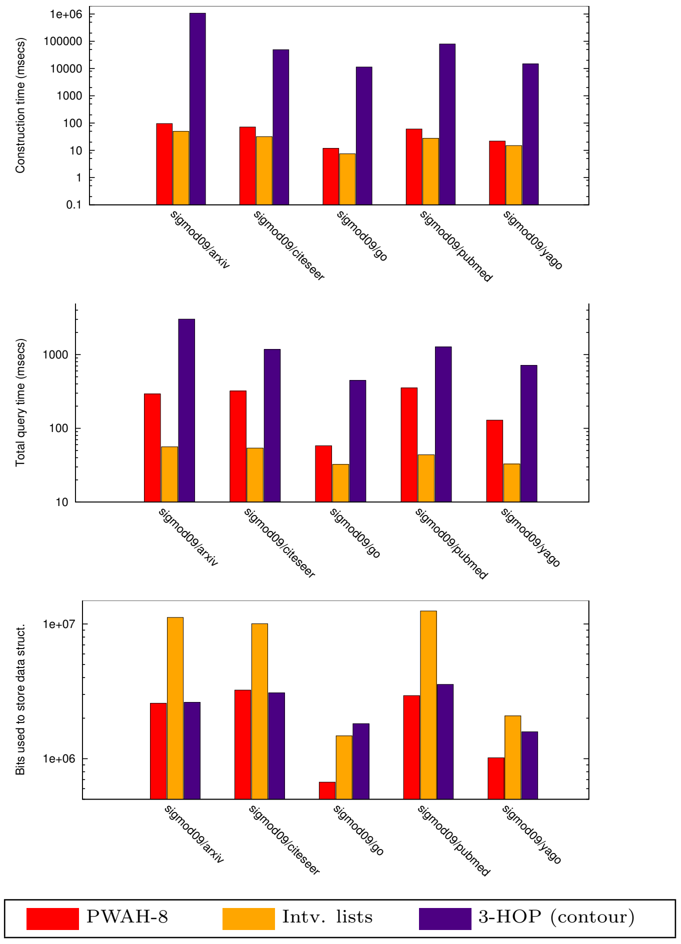


Figure 11: Performance comparison: PWAH-8, interval lists and 3-HOP

to process the graphs. The extended construction time does not result in an improved query time or smaller memory footprint: both PWAH-8 and interval lists provide a better query time, with interval lists taking the lead. In terms of memory usage, PWAH-8 performs surprisingly well and generally significantly better than interval lists and 3-HOP.

7. CONCLUSIONS

We have demonstrated how the use of bit vector compression leads to a highly memory-efficient data structure for representing reachability information. Based on the characteristics of this application, we proposed a small improvement to the well-known word-aligned hybrid (WAH) compression scheme, named partitioned word-aligned hybrid compression (PWAH).

In a theoretical analysis, we showed that PWAH-8 outperforms interval lists (a straightforward but quite effective way of achieving compression originally suggested by Nuutila [11]) in terms of memory usage.

We conducted experiments on a wide variety of graphs, both synthetic and from real applications. These benchmarks include the graphs proposed by others who have been working on the same problem [9, 8], as well as new examples that are several orders of magnitude larger.

These experiments indicate that in terms of memory usage, the new technique of compactly representing reachability improves on all previous algorithms. In particular it allows us to process much larger graphs than those handled by other recent proposals such as Path-Tree [9] and 3-HOP [8]. We used the implementations provided with those papers for our experiments.

The new data structure also performs very well in terms of query time. In some cases, however, the query time of interval lists is somewhat better. We feel this is an appropriate trade-off since memory usage is the primary obstacle to computing reachability information in truly large graphs.

There are two directions for further research that we are now pursuing. The first is to investigate more deeply the performance of PWAH and interval lists on yet larger graphs. On the largest example considered here, it appears that PWAH is better both in terms of memory usage and query time but it is unclear whether that would be the case for other inputs of comparable size. The second direction is to explore applications in program analysis: our experiments show that the new data structure enables analyses that were hitherto unthinkable. We are also exploring potential applications in semantic web technology.

In this paper we have assessed the performance of PWAH exclusively for the application to compressing reachability information. It would be interesting to explore whether other applications of WAH could benefit by the introduction of partitions.

8. REFERENCES

[1] R. Agrawal, A. Borgida, and H. Jagadish. Efficient management of transitive relationships in large data and knowledge bases. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, pages 253–262, 1989.

[2] R. Agrawal and H. Jagadish. Direct algorithms for computing the transitive closure of database relations. In *13th VLDB conference*, pages 255–266, 1987.

[3] E. Cohen, E. Halperin, H. Kaplan, and U. Zwick. Reachability and distance queries via 2-hop labels. In *Symposium on Discrete Algorithms*, pages 937–946, 2002.

[4] J. Eve and R. Kurki-Suonio. On computing the transitive closure of a relation. *Acta Informatica*, 8:303–314, 1977.

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

[6] V. Hirvisalo, E. Nuutila, and E. Soisalon-Soininen. Transitive closure algorithm MEMTC and its performance analysis. *Discrete Applied Mathematics*, 110(1):77–84, 2001.

[7] Y. E. Ioannidis, R. Ramakrishnan, and L. Winger. Transitive closure algorithms based on graph traversal. *ACM Transactions on Database Systems*, 18(3):512–576, 1993.

[8] R. Jin, Y. Xiang, N. Ruan, and D. Fuhry. 3-HOP: a high-compression indexing scheme for reachability query. In *SIGMOD '09: Proceedings of the ACM SIGMOD international conference on Management of data*, pages 813–826, New York, NY, USA, 2009. ACM.

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

[10] E. Nuutila. An efficient transitive closure algorithm for cyclic digraphs. *Information Processing Letters*, 52:207–213, 1994.

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

[12] E. Nuutila and E. Soisalon-Soininen. Efficient transitive closure computation. Technical Report TKO-B113, Helsinki University of Technology, Laboratory for Information Processing Science, 1993.

[13] P. Purdom. A transitive closure algorithm. *BIT Numerical Mathematics*, 10(1):76–94, 1970.

[14] T. 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.

[15] T. Reps, S. Horwitz, and M. 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.

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

[17] L. Schmitz. An improved transitive closure algorithm. *Computing*, 30(4), 1983.

[18] K. Stockinger, J. Cieslewicz, K. Wu, D. Rotem, and A. Shoshani. Using bitmap indexing technology for combined numerical and text queries. In *New Trends in Data Warehousing and Data Analysis*, volume 3 of *Annals of Information Systems*, pages 1–23, 2008.

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

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

[21] K. Wu et al. Word-aligned hybrid compression for bitmap indices. Website with overview of applications `http://crd.lbl.gov/~kewu/fastbit/compression.html`, 2010.

[22] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database Systems*, 31(1):1–38, 2006.

[23] L. Zhu, B. Choi, B. He, J. X. Yu, and W. K. Ng. A uniform framework for ad-hoc indexes to answer reachability queries on large graphs. In *Database Systems for Advanced Applications*, volume 5463 of *Lecture Notes in Computer Science*, pages 138–152, 2009.