

Efficient data structures for sparse network representation

Jörkki Hyvönen*, Jari Saramäki, and Kimmo Kaski

Laboratory of Computational Engineering, Helsinki University of Technology, P.O.Box
9203, FI-02015 TKK, Finland

(Release date)

Modern-day computers are characterized by a striking contrast between the processing power of the CPU and the latency of main memory accesses. If the data processed is both large compared to processor caches and sparse or high-dimensional in nature, as is commonly the case in complex network research, the main memory latency can become a performance bottleneck. In this Article, we present a cache efficient data structure, a variant of a linear probing hash table, for representing edge sets of such networks. The performance benchmarks show that it is, indeed, quite superior to its commonly used counterparts in this application. In addition, its memory footprint only exceeds the absolute minimum by a small constant factor. The practical usability of our approach has been well demonstrated in the study of very large real-world networks.

Keywords: Complex network, Hashing, Linear probing, Data structure, Cache

1. Introduction

1.1. *Complex networks and their representation*

Over the past decade, it has become apparent that *networks* are an efficient way to represent and study a large variety of complex systems, including biological, technological and social systems (for reviews, see, e.g., [1–3]). The strength of the network approach lies in its inherent ability to discard unnecessary details and capture the essential structural properties of complex systems. In the complex networks framework, interacting elements are represented by the nodes of a network, and their interactions by edges connecting the nodes. It has been discovered that there are structural characteristics common to a very large number of networks, such as short path lengths (the "small-world" property) and broad connectivity distributions where a small fraction of vertices (the "hubs") have an exceedingly large number of connections.

There are several computational challenges associated with empirical network analysis and network simulations. Networks are typically very sparse, high-dimensional structures. Mathematically, a network can be represented by the *adjacency matrix* A , where the element $a_{ij} = 1$ if vertices i and j are connected, and zero otherwise. In case of weighted networks, an edge is associated with a weight. This can be captured by the *weight matrix* W , where w_{ij} represents the weight of the edge connecting i and j ($w_{ij} = 0$ signifies the absence of the edge). For undirected networks, $A = A^T$ and $W = W^T$. An example of a simple undirected network and the corresponding adjacency and weight matrices are shown in Fig.

*Corresponding author. Email: jjhyvone@lce.hut.fi

1 a). It is evident that simply due to memory consumption, such matrices cannot be directly used as data structures for encoding networks, especially when large networks are considered. The high dimensionality of the data provides a further challenge. Typically, empirical analysis of networks involves extracting statistical quantities related to vertices or edges and their immediate neighbourhoods; some analysis methods probe larger-scale structures such as densely connected communities or percolating clusters. These methods may also involve restructuring of the network by removing edges or permuting the structure, and hence in addition to minimizing the memory footprint, the utilized data structures should provide fast means for traversing, adding, and deleting nodes and edges. This requirement is especially crucial for analysis of networks where the number of vertices $N \sim 10^6$ and number of edges $E \sim 10^7$ (see, e.g., [4, 5]). A visualization of a part of such a network is shown in Fig. 1 c). Large-scale simulations are also demanding – the typical approach is to generate a large statistical ensemble of simulated networks for extracting meaningful statistics, or to average over a large number of runs of simulated dynamics.

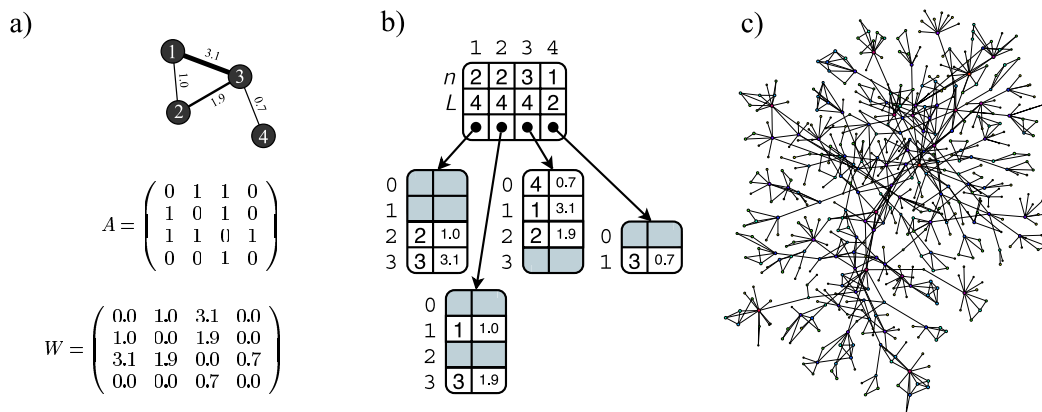


Figure 1. a) A small 4-node network and the associated adjacency and weight matrices. b) The network implemented as an array of static parts of the data structures representing edge sets, that is, the vectors of the adjacency matrix. The edge sets are hash tables with $h_L(k) = n \bmod k$. Collision resolution is not necessary in this simple example. See the Section 3 for details. c) Visualization of a small ($N = 533$) sample of the large social network studied in Refs. [4, 5], illustrating the complexity inherent in network data.

1.2. Computer memory and caches

It is well-known that during last few decades, the raw computational power of microprocessors has grown approximately exponentially, as predicted by Moore's law. However, less attention has been drawn to the fact that the performance of memory chips, measured as *latency* between a load request and the arrival of the corresponding data into the *registers* of the processors, has seen very little improvement. Indeed, the main memory latency can be as high as several hundred clock cycles of modern CPU:s. The *superscalar* operation of processors, allowing them ideally to finish multiple instructions per clock cycle, makes the situation even worse: as many as a thousand machine code instructions could have been finished within the time needed for the data to be processed to arrive at the registers.

Modern computer architectures tackle this apparent bottleneck essentially in two ways. *Caches* have been introduced between the processor and the main memory. They comprise of memory significantly faster but more expensive than the main memory itself. Recently accessed data will be kept in the cache. This approach relies on *temporal locality*, which means that the data most probably accessed in

the future is that accessed in the recent past. In modern day processors, the caches come in a hierarchy of two to three levels of increasing size and latency. They can do an impressive job; for example, simulations [6] show that *cache hit rates* of common CPU benchmark programs are typically as high as 99%.

In another effort to counter the high latency of the main memory, its *bandwidth* has been increased steadily. Essentially, many individual memory modules corresponding to adjacent locations are activated simultaneously and their content is sent to the processor through an extremely fast bus. In order to exploit this, caches are organized in *lines* of equal length — typically, 32 or even 64 bytes. Single memory request leads to the loading of the whole cache line. The phenomenon exploited here is that of *spatial locality*: it is probable that the access of a data element is followed by that of an adjacent one in the near future — perhaps a member of the same *struct* or *object*, or another element in the same vector. Different kinds of *prefetch* logic are utilized by processors. For example, two cache misses corresponding to successive memory areas cause automatic loading of a third, immediately following area in the cache. If the data is processed serially, as in the case of data streams occurring in multimedia applications or in matrix algebra involving long vectors, the memory latency bottleneck can be practically eliminated. Depending on the cost of processing the loaded data, the computation can even become *bandwidth-limited*.

The memory latency becomes a problem in cases where the data processed is very high-dimensional or sparse, but large. This is the common case in network science. Such data cannot be laid out in memory in such a way that conceptually adjacent elements — such as connected nodes in a network — would also be adjacent in the main memory. There is thus no spatial locality to be exploited by the caches. Of course, temporal locality depends on the exact nature of processing. However, when large data sets are considered, typical operations, such as following edges on a path in a network or finding common neighbours of nodes are, on the average, bound to access non-cached memory. As cache misses are thus unavoidable, the relevant problem becomes that of finding data structures to minimize their number.

At this point, it is helpful to point out the similarity of the distinction between fast caches and slow main memory to that between main memory and mass storage. Indeed, modern hard disks have rather long *seek times* — of the order of several milliseconds at best — but the data transfer rates can be as high as a hundred megabytes per second. Modern operating systems cache disk accesses: the equivalent of a line of a processor cache is a *memory page* which is the smallest amount of data read into main memory when the disk is accessed, typically a few kilobytes in size. On the other hand, not too long ago the amount of main memory available on computers was so low that it was common to only load a small part of the data to be processed into the memory, most of it only being stored on a disk. Data structures found efficient for disk-based processing can therefore provide us with a good starting point.

2. Implementation basics

As discussed above, networks and their representative adjacency or weight matrices are mostly very sparse; the network approach to a problem might not be fruitful otherwise. Thus our task of designing an efficient data structure for encoding networks partially reduces to finding an efficient representation for sparse matrices. The nodes of a network — corresponding to rows and columns of the matrix representation — can usually be laid out continuously in the memory as an array. The indices of nodes only serve as labels, which can always be chosen as integers

from the interval $0 \dots N - 1$, where N is the number of nodes in the network. The same is clearly not true for the *content* of these vectors, implicitly containing the indices of adjacent nodes and explicitly e.g. the weights of edges. Thus, finding an efficient implementation for these vectors associated with nodes and representing their connections is of our main concern.

The efficiency of the implementation is measured by the memory footprint of the data structure and the processor time needed for individual operations. Due to spatial locality, these metrics are often closely related. In addition to usual network modifications, i.e. removals and additions of edges, there are two operations of great importance, namely *seek* and *iteration*. The former corresponds to checking for the existence and perhaps the weight of an edge, that is, an element of the adjacency or weight matrix, whereas the latter corresponds to going through all edges connecting to a given node. As the adjacency matrices of networks are very sparse, the usual approach of going through all N potential elements of their vectors is doomed to be extremely inefficient. Due to the broad connectivity distributions, almost all elements are zero for most of the nodes, corresponding to non-existent edges. When performing standard matrix algebra, basic operations such as addition and multiplication can exploit the sparsity “inside” the data structures, as the algorithms for iteration are not visible to the user. However, this is not true for network science in general: many commonly used algorithms are simply not reducible to these operations, but call for explicit iteration over the neighbours of nodes.

In the following, we consider the *hash tables* which can be used to represent the node-specific vectors.

3. Hash tables

Typical *linked* data structures such as trees [7] are simply not efficient enough for our purpose, as traversing them either during seek or iteration involves repeated following of *pointers*, usually referring to memory that is not loaded into the processor cache. In addition, there is an *allocation overhead* for each node, typical value of which is two machine words [8] — that is, 64 or 128 bits — per node. The main virtues of trees, ordered iteration and small worst-case access times, are not very helpful for handling networks. Ordering of edge sets is unnecessary as the node indices only serve as labels, though it can be exploited, for example, in order to make sure that each individual edge of an *undirected* network is iterated over just once. On the other hand, as we are not interested in real-time computation, it is only the average time needed for different operations that is of interest to us.

Thus, as a starting point, we choose to lay our data comprising of node indices — implicit in a non-sparse vector, but explicitly needed here — and possibly corresponding edge weights contiguously in memory. The whole network would then be implemented as an array of nodes, each containing a pointer to the corresponding memory area and the number of edges in it.

Iteration over packed array edge lists is extremely fast, as many elements can usually fit on a single cache line, and the total number of cache misses during the whole operation is bound by the prefetching logic of the processor. If the elements are stored sorted according to their indices, the seek operations can be performed by interval halving, leading to $O(\log n)$ time requirement, where n is the number of non-zero vector elements. This is a common form used by eg. MATLAB® for sparse vectors. However, additions and removals of elements require displacing half of the array, on the average, thus leading to $O(n)$ cost. If we chose to store the elements unordered, additions could be performed in constant time, but $O(n)$ seek

time would result.

3.1. Hash functions

In general, these kinds of data structures are referred to as *sets* and *maps*. In our application, the *keys* stored in the sets are just the indices of non-zero elements in the vectors of the adjacency matrix, that is, the identity of the neighbouring nodes. In the case of weighted networks, the values the keys map to are the edge weights. In the following, we refer to keys or combinations of keys and values simply as *elements*.

A common solution for efficient implementation of sets and maps is the utilization of *hash tables* [7]. Let us take a contiguous array in the memory capable of holding L elements accompanied by their indices. In addition, a way of marking each *slot* of the array either used or unused is needed. When adding an element into this table, a *hash function* $h_L(k)$ mapping each key to an integer on the interval $[0, L - 1]$ is utilized to calculate the slot for it. A simple example would be a modulo operation: $h_L(k) = k \bmod L$. A network data structure based on hash tables corresponding to the network of Fig. 1 a) is shown in Fig. 1 b).

3.2. Collision resolution

If the keys were known beforehand, it would be possible to construct a hash function that would map each key into a different slot in the table, even if $L = n$. In practice, this requirement is certainly not met, and different keys may map on a same slot. Some form of *collision resolution* is thus needed.

A simple and perhaps the most commonly utilized collision resolution method is *chaining*, exemplified in the Fig. 2 a). In this strategy, a slot of the table is not used to hold the keys and values themselves but a root pointer to a linked list which holds the individual elements belonging to the slot. It is rather easy to show that given random keys, chaining results in $O(1)$ average seek cost, if $L \propto n$, even with very simple hash functions. On the other hand, it is always possible to pick a set of indices in such a way that they all map onto the same slot. Choosing a hash function that produces approximately even distribution of the keys on the slots given the possible regularities in the key set is thus of great importance.

For our application, the chaining strategy fails because of the additional latency involved in following the link pointers on the list. During a seek operation, a small additional number of cache misses might be quite tolerable. However, iteration through elements is very slow compared to a sequential memory access, which might only suffer from one or two cache misses during the whole iteration.

3.3. Open addressing

Another common form of collision resolution is *open addressing*, which utilizes the table itself to store the elements that have their initial slot given by the hash function already occupied. In general terms, a sequence of successive hash functions $h_{1,L} \dots h_{L,L}$, each mapping the same key on a different slot on the table is utilized. For a given key, these functions define a *probe sequence* spanning the whole table. A search operation is carried out by applying each hash function successively on the index and checking the corresponding slot until the key is either found or an empty slot is encountered. When adding a new element into the table, it is inserted into such an empty slot. However, removals of elements are a bit tricky in general. The slot from which an element is removed might be a preferred one for some other key,

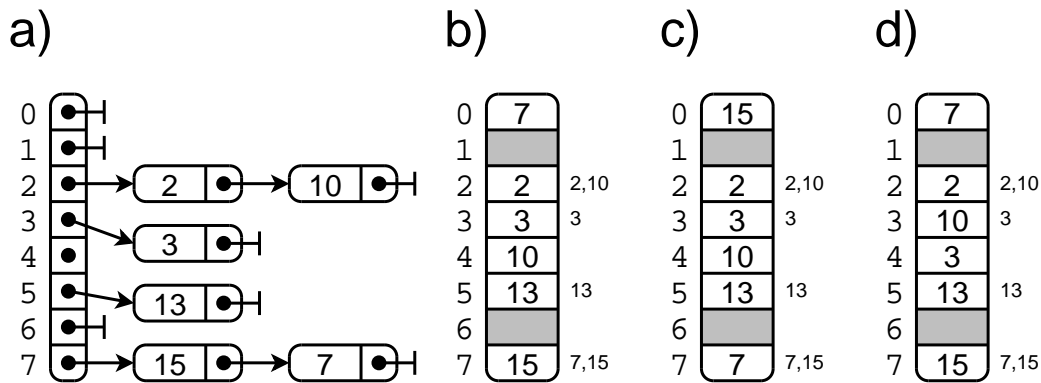


Figure 2. Hash tables with different form of collision resolution. Hash function is $h_g(k) = k \bmod 8$. In the tables utilizing linear probing, the initial slots for each key are shown beside the tables. Key 15 is inserted before the key 7 and keys 2, 3 and 10 in this order. a) Collision resolution by chaining. Newly inserted elements have been inserted at the tails of the linked lists. b) Linear probing, First come-strategy. Elements are inserted in the first empty slot found in the probe sequence. c) Linear probing, ordered hash table. Compared to b), the keys 7 and 15 are now ordered by their numerical value. d) Linear probing, Robin Hood strategy. The keys 7 and 15 are in their order of insertion, whereas 10 and 3 are both displaced by the same amount from their initial slots.

already in the table. Simply marking the slot as unused would then cause a “hole” in the probe sequence corresponding to another key, thus terminating a search for it prematurely. A special mark is thus utilized for slots from which an element has been removed. Such a slot is then treated as a used one during seeks and as an empty one during insertions.

The chaining hash table implementation suffers only from a graceful degradation of performance when the number of elements n exceeds the number of slots L . In general, however, the table should be *rehashed* into a larger size when the average number of elements per slot reaches some given constant. During rehashing, each element is simply inserted into a new table instance. In order to save memory space, rehashing to a smaller size can be performed when the number of elements becomes low. However, this operation can be postponed to certain extent e.g. when in a performance-critical section of a program. In the case of open addressing, the limit for the number of elements set by the length of the table is quite hard, the performance degrading significantly when the *fill rate* $\alpha = n/L$ of the table gets near 1.¹ For keys randomly distributed in the table, the seek time still only depends on n and L through α . Thus, for a constant α the seek cost is always $O(1)$, that is, independent of the number of elements or the size of the table.

It is easy to prove by *amortized* analysis [9] that the average time needed for addition of elements is $O(1)$ even if rehash operations are needed. Intuitively, this can be seen as follows. Let us consider the case in which the table size is doubled when a limit on the fill rate is reached. Just before a rehash, half of the elements in the table have been inserted in the table just once, after the previous rehash operation. Half of the remaining elements have been only rehashed once after their initial insertion, half of the other half just twice etc. The series converges in such a way that the total number of insertions into a table of some size is two times the number of elements in the final one.

¹The n in the expression for the fill rate should include the elements marked as removed, also.

3.4. Linear probing

For spatial locality, we choose to utilize the simplest possible form of open addressing, which is *linear probing*. The probe sequence is simply such that $h_{1,L}(k) = h_L(k)$ and $h_{n+1,L}(k) = (h_{n,L}(k) + 1) \bmod L$. Successive slots in the table — wrapping around to 0 at L — are thus probed until a key is found or an empty slot is encountered. An example of a hash table utilizing linear probing is shown in Fig. 2 b). In theory, linear probing is less efficient than more elaborate probing schemes when the fill rate α is near 1. This is due to *clustering* in which populated areas in the table tend to join together when the fill rate is increased. [7]. The fact that successive probes are very likely to fall in the same cache line is expected to more than compensate for this handicap. Some experimental proof for this exists in [10]. Interestingly, linear probing was seen as the preferred solution for *external searching* for data on mass storage by Donald Knuth in [7]. It is worth noting here that the asymptotic time requirement for seek operations of full tables for which $n = L - 1$ is $O(\sqrt{n})$ for successful and $O(n)$ for unsuccessful seek operations [7].²

Because the probe sequence in linear probing is solely determined by the initial slot for a key $h(k)$, key removal can be performed efficiently. If a key is removed, the sequence of following slots is inspected for keys eligible for insertion into the newly freed slot. If none are found before the next empty slot, then it suffices to mark the slot of the removed element as free. If an eligible key is found, it is moved into the freed slot and the process is repeated for its original slot. In this way, there is no need to maintain a separate count for slots marked as removed, and performance degradation — and thus, rehash operations — are not caused by removals.

3.4.1. Linear probing variants. Above, we have considered the “greedy” or *First Come* form of linear probing, in which a slot is occupied by the first eligible key inserted in the table, following ones placed further in the probe sequence. [7, 11] Other alternatives maintaining the validity of probe sequences exist. Most obviously, insertions may be done in *Last Come* –fashion, new key k replacing an old one in its initial slot $h(k)$ [11]. It is also possible to maintain the keys in a table in a sorted order, a key with the smallest numerical value taking precedence on a slot when resolving collisions [7, 12]. The interesting consequence of this strategy is that an unsuccessful search can be terminated as early as the corresponding successful one. Indeed, the slot allocation of a sorted hash table is unambiguous for any given set of keys, and a seek operation can terminate when the probe sequence passes the slot in which a key would be placed, if present. Fig. 2 c) shows an example of an ordered hash table.

It is seldom useful to utilize very large fill rates. The average increase in the memory footprint of hash tables is only slight when the fill rate — or, in practice, the rehash limit of the fill rate — is increased from, say, 0.8 to 0.9. The degradation of performance is large for such an increase, however, as seen from the results in Ref. [7]. A better reason for cutting down the asymptotical cost of searches is the clustering phenomenon, which can make the performance more sensitive to regularities in the data. Chaining strategies only suffer if keys are mapped to exactly same slots. In the case of linear probing, tendency of keys to map on same *regions* of the table can amplify clustering. Utilizing the ordering to minimize the cost of unsuccessful searches can thus be useful even with more sensible fill rates.

It should be noted that the average length of probe sequences for all keys in a table is the same under any form of linear probing. This is easily seen by considering

²This case does not correspond to any constant α .

swapping of any two keys that are eligible for insertion in each other’s slots. The swapping maintains the sum of the lengths of the probe sequences for the keys or alternatively, the sum of their displacements from their initial slots.

An interesting collision resolution strategy is *Robin Hood*, in which the key k to take the precedence of a slot is the eligible one with the largest displacement from its initial place $h(k)$. Rather intuitively, this is the strategy that minimizes the variance of the length of the probe sequences keys in a table [11, 13, 14]. This property can be useful in modern microprocessors. Their *pipelined* operation is very sensitive to *branch prediction* of conditional jumps in the code. [15] If the processor is able to guess the program flow across the conditionals, processing is uninterrupted. On the other hand, *mispredicted* branches cause considerable delays in processing. Minimizing the entropy of the probe sequence length distribution should help the branch prediction hardware to work more efficiently.

Although not noted in the cited literature, it is possible to utilize the “ordered” nature of Robin Hood hashing to cut down the cost of unsuccessful seek operations, as in the case of ordered hashing. The comparison of numerical key values is replaced by that of the displacement of keys from their initial slots. However, the allocation of the keys into the table is not unambiguous as in the case of ordered hashing — exactly same keys in the same table are not allowed in our application, but the initial slots for different keys can of course be same. The performance is thus expected to be slightly worse than in the case of ordered hashing proper: all the keys mapping to the same initial slot have to be passed before a seek can be terminated as unsuccessful.

4. Results

The main problem in establishing relevant performance figures for data structures intended for edge set implementation is the access of non-cached memory. The building of an edge set for a performance measurement itself results in the structure being cached. Thus, in general, one would have to construct a large number edge set instances in order to drop the least recently accessed ones out of the caches, and then perform experiments on those. However, the exact form of the memory allocation strategies used by the standard libraries might result in artificial spatial or temporal locality affecting the results.

At least for the random sets of keys used in our experiments, all hash table implementations have seek times with no direct dependence on their size. It is thus possible to simulate uncached seek operations on small tables by those on a single large hash table instance with the same fill rate. The keys have to be accessed in a sufficiently random order so that the effect of spatial and temporal locality is negligible. Gradually decreasing the size of the tables used in measurements will then serve as the visualization of the effect of caching.

4.1. Benchmark setup

The seek operations of the linear hash tables were benchmarked for tables in sizes ranging from 16 (2^4) to 268435456 (2^{28}). For each size, fill rates from $7/16 \approx 0.44$ to $13/16 \approx 0.81$ with spacing $1/16$ were experimented on. For the largest table size considered, only single hash table was used and the number of seek operations made was equal to the number of keys in the table. For tables of decreasing size, the number of tables used and that of seek operations per element in the table were gradually increased so that the total number of seeks per each measurement

point was constant for a given fill rate. Furthermore, all the measurements were made for both successful and unsuccessful searches.

Here we did not run any benchmarks on addition and removal of keys. This is due to the fact that the cost of both of these operations reduces to the cost of an unsuccessful search from an unordered hash table: first, the key or the place for its insertion is found, and then all the following ones until the next vacant slot are moved by a single slot.

The table implementation utilized was a set in which the usage status of a slot was marked by a special “magic value” of the key. Both First Come and Robin Hood collision resolution strategies were used in the experiments. In order to achieve realistic performance measurements, the multiplicative Fibonacci hash function [7] was utilized, although even a simple modulo operation would have performed as well for random keys used, and would have been extremely efficiently implemented by bit-masking for the table sizes used here.

For comparison, similar benchmarks were run on the set structure of the C++ standard library based on a red-black tree [16] and the set container based on a chaining hash table found as an extension to the GNU C++ library. The latter is a direct adaptation of the hash table of the Sun Microsystems Standard Templates Library (STL). [17] The fill rates used for it varied approximately from 7/12 to 1 for each table size considered.¹ Of course, the fill rates are not directly comparable: the storage efficiency of the linear probing table is much better than that of the chaining one, due to the space required by pointers and node allocation overhead. For the red-black tree, the number of elements used in the benchmarking were simply successive powers of 2.

With one exception to be explained later, all the benchmarks were run on a 3 GHz Pentium 4 platform having 3 GB of memory. The operating system was Gentoo Linux with kernel version 2.6.9. The GNU C++ compiler version 3.3.5 was used with the simple -O as the sole optimization parameter.

4.2. *Benchmark results*

4.2.1. Successful searches. The results for successful searches are shown in Fig. 3 for the linear probing hash table, the STL hash table and the set implementation of the C++ standard library. As seen in the figure, the linear hash table performs better for successful seek operations throughout the range of set sizes considered. The small seek times of the chaining STL hash table for very small set sizes can be accounted to the rather large minimum size of the table. The higher memory efficiency of the linear implementation is reflected by the fact that the steep rise of the seek time corresponding to the structure not fitting in the level 2 of the processor cache, occurs considerably later than for the STL table. The performance of the Robin Hood implementation is somewhat less than that of the First Come. This is due to the additional code used for checking the ordering of the elements, which might be ineffectively optimized by the compiler.

4.2.2. Unsuccessful searches. The results for unsuccessful seek operations are shown in Fig. 4. In this case, the dependence on the fill rate is very strong for the linear hash table with First Come hashing. With the highest rates considered, the seek time even exceeds that of the chaining implementation. If the maximum

¹The table sizes used in this implementation are primes close to 2/3 time successive powers of two. The smallest table size available is 53

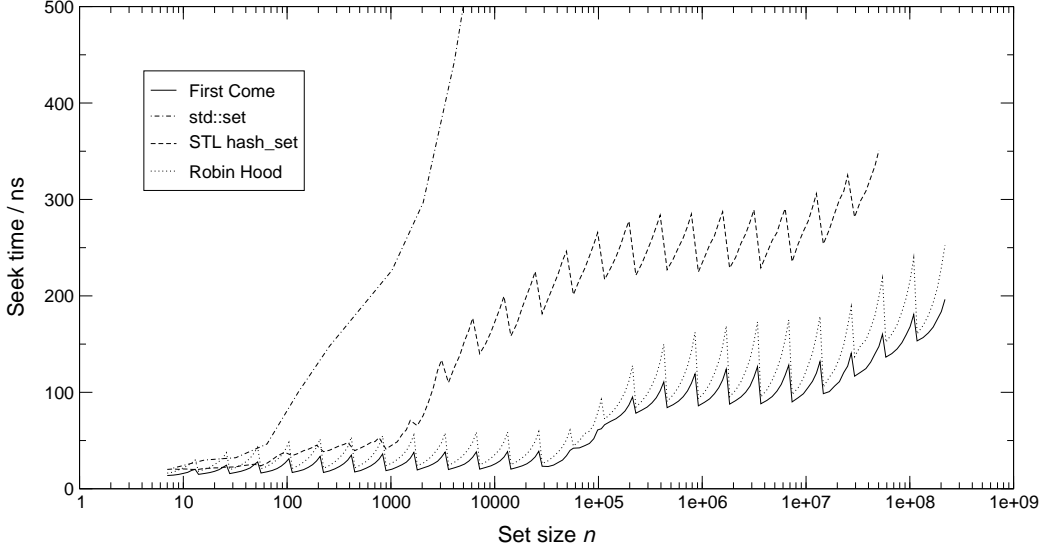


Figure 3. The average times needed for successful seek operations for the data structures considered. The characteristic sawtooth pattern resulting from the variation of the fill rate is clearly seen for the implementations based on hash tables. The sudden rise of the seek times near the center of the graph results from the largest tables not fitting in the level 2 of the processor cache. The set implementation of the standard library is based on a tree. The time complexity of the seek operations should therefore be logarithmic and the corresponding curve to be linear on the semi-logarithmic plot. The regions of different steepness again correspond to the processor cache: in this case, the effect of the 1st level cache is also visible. The rise of the curves towards the highest set sizes considered can be accounted to memory management issues of the operating system.

fill rate were lowered in such a way that the average memory consumption would be equal to that of the chaining implementation, the linear probing hash table would again emerge as a clear winner. In any case, the difference in the results between successful and unsuccessful seek operations is in a nice agreement with the theoretical predictions presented in Ref. [7]. As expected, the Robin Hood strategy scores quite a lot better than First Come for unsuccessful seeks.

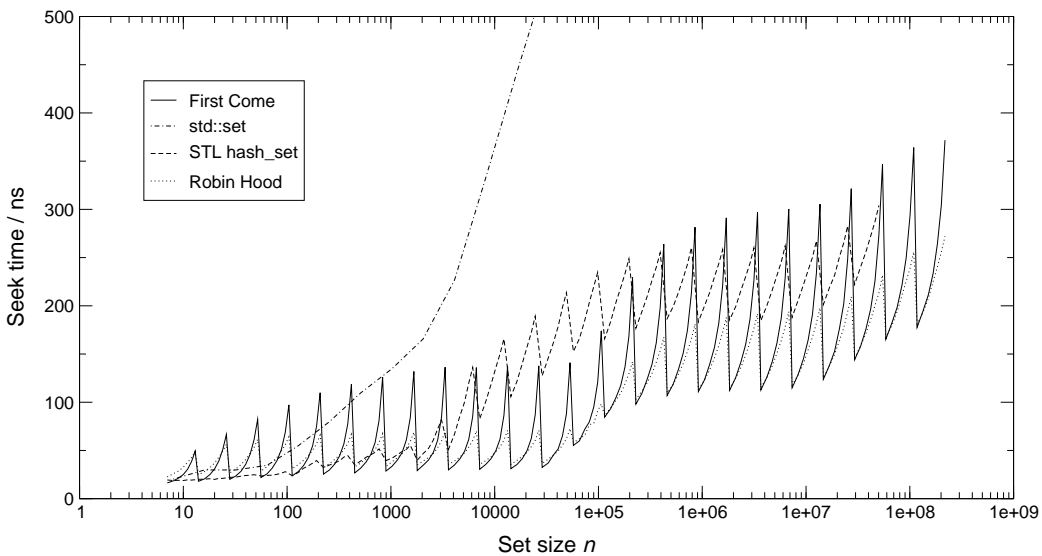


Figure 4. The average times needed for unsuccessful seek operations from the data structures considered. The high variation of the seek time of the First Come table with its fill rate is clearly visible. The Robin Hood variant fares considerably better in this respect.

4.2.3. Prefetching. In the linear probing hash table, it is highly likely that a key is found or the search can be terminated on the same cache line as its initial slot. It is thus possible to utilize *prefetch* instructions of modern processors to load the memory area corresponding to the initial slot of a key into the cache, if the key to be accessed is known suitably beforehand. Ideally, this approach can mask out the memory latency altogether. Chaining implementations would not benefit very much from prefetching: although the slot of an element is always prefetched correctly, the target of the pointer therein can certainly not be prefetched until the data is in the cache.

The `__builtin_prefetch` instruction of GCC, translated into the corresponding instruction of the SSE extension of the machine language was used. The prefetch instructions were issued 6 seek operations beforehand. Unfortunately, the generation of Pentium 4 processor that was used for all the other benchmarks has the rather well-known feature that prefetches of memory locations that cause a TLB cache miss are ignored. No real performance improvements could therefore be seen on this platform. A computer with an Athlon 64 FX-51 processor and 1 GB of memory was employed in the prefetch benchmarks in order to obtain meaningful results. The computer ran the 64 bit version of the Mandrake Linux; thus the machine word size was double that in the principal benchmark platform. Successful searches were performed with the similar procedure as in all other seek benchmarks. The results are shown in Fig. 5. The overhead associated with prefetching is very small, as seen in the left side of the figure corresponding to sets fitting entirely in the processor cache. This overhead also contains the additional evaluation of the hash function required for prefetching. For the largest set sizes, the prefetching significantly improves the performance. The exact size of the effect remains unclear, however, as the finite amount of the available system memory starts to affect the results at rather small set sizes. The corresponding results on the primary test platform were not included in the figure in order to improve the readability: below the cache size limit, they were very similar to those presented.

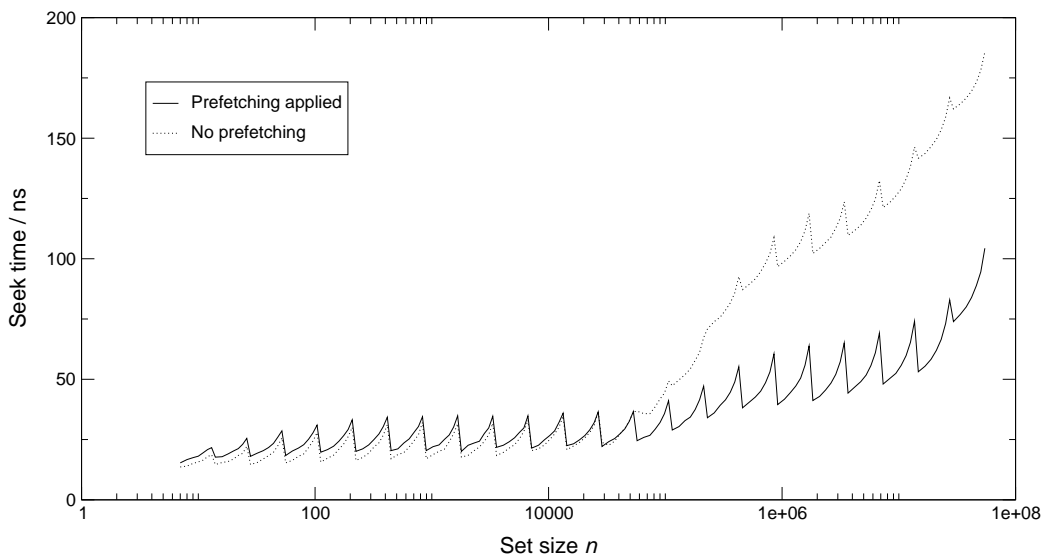


Figure 5. The average time required for normal and prefetched successful seek operations. Prefetches were issued 6 seek operations beforehand: results were, however, very insensitive to the exact number used. See the text for details.

In practice, prefetch instructions can be efficiently utilized for example when calculating the intersection of two sets, i.e. the set of common neighbours of two nodes. The smaller set is iterated over and each element is searched from the larger

set. The same method would apply to the calculation of sparse vector dot products.

4.2.4. Iteration. The time needed for iterating over the elements of STL hash sets and linear probing hash tables of different sizes is shown in Fig. 6. The procedures employed in the iteration benchmark are similar to those used in the seek tests. The keys found during the iteration were counted to prevent the compiler from optimizing away parts of the otherwise effect-free procedure. For large enough tables, the linear implementation is an order of magnitude faster than its chaining counterpart. If a large number of very small sets were considered, cache miss latency for the table itself should be added to the time demand of both the implementations.

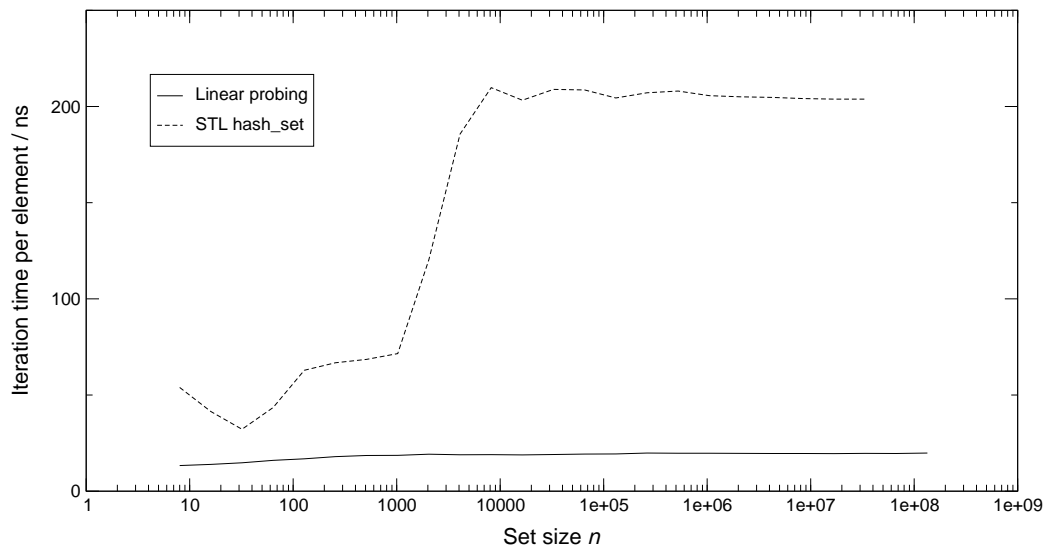


Figure 6. The average time required for iterating over a single element in the chaining and linear hash tables. The memory latencies induced by pointer traversals render the chaining implementation slow for large table sizes. The large minimum size of the table, in turn, causes the slowness in the limit of very small sets.

4.3. Comparison with other implementations

In [18] Donald Knuth presents a graph library based on representation of edge sets as linked lists. Though elegant in implementation, this approach is handicapped by $O(n)$ access time, node allocation overhead and the high cost of following a pointer into non-cached memory.

The IGraph library by Gabor Csardi [19] includes an interesting sparse network implementation. The target nodes of *all* edges in the network are stored in a single array, as are the corresponding weights. An additional array stores the location of edges associated with each node in them. The memory footprint of this kind of implementation is truly minimal. Seek operations can be performed in logarithmic time, as the edge sets are stored as ordered. Iteration over the neighbours of a node is very fast, as only sequential memory access is needed. However, a major problem with this implementation is the efficiency of adding edges into the network. All the following edges in the huge arrays have to be displaced in order to make room for a new one. The time requirement of such an operation is thus $O(E)$, where E is the *total* number of edges in the network. Hence, the cumulative cost of constructing a network with E nodes is $O(E^2)$, unless nodes and edges are added in a strictly ordered fashion.

Keith Briggs has published a sparse network implementation [20] based on the chaining hash table of GLib [21]. It has the interesting feature that all the edges in a network are stored in a single table. Ideally, accessing a given edge could be very fast, as only one access of possibly non-cached memory is needed. However, following at least one additional pointer is needed because of the chaining implementation of the hash table. In addition, there is no efficient way of iterating through the neighbourhood of a node in this implementation.

4.3.1. Memory footprint. The space efficiency of an implementation places a fundamental limit on the size of the networks that can be studied efficiently. This limit is quite hard because, in general, complex network simulations cannot be efficiently distributed over several distinct memory spaces in e.g. a networked computer cluster. This is due to the inherent high dimensionality of networks: a high proportion of edges would inevitably be between nodes in different memory spaces. Following such edges would then cause a latency orders of magnitude larger than that of a cache miss.

In Table 1, the per-node and per-edge memory requirements of several network implementations are presented. It has been assumed that both the indices of nodes and the weights of the edges are represented by a single machine word. In addition, the dynamic memory allocation overhead is assumed to be two machine words as in [8], but in such a way that all sizes of memory chunks can be allocated without waste of space. In the array-based implementations, the allocation overhead of the array is included in the node’s memory requirement. Thus, nodes with no associated edges would in reality require two machine words less memory.

In addition to the implementation based on the linear probing hash table and the ones by the other authors mentioned above, two hypothetical implementations are included in the Table 1. In the first one, the edge sets are packed arrays as described in the Section 3, and in the second they are STL hash tables used in the benchmarks of section 4.

5. Conclusion

By utilizing cache-aware data structures, we have been able to introduce a network representation that performs very much better than data structures found in common programming language libraries. In addition, the memory footprint of the basic data structure used, i.e. the linear hash table, is very low. The static part of the structure that is stored in the node vector only consists of a pointer, number of elements, and the table size. The first two are essentially identical to the “minimum” implementation of edge set as a packed array. The last, in turn, can be made to occupy only a few bits. This is because we can constrain ourselves to table lengths of powers of two, and thus only the power — the 2-based logarithm of the table length — has to be stored.¹ The dynamic part of the structure containing the edges differs only from the absolute minimum — the space required for the storage of the target node and weight — by the factor of average fill rate α . In fact, the difference can be even smaller: the table sizes are powers of two, and practical memory allocators such as the one presented in Ref. [8] can usually provide these sizes without any waste of space.

¹In fact, Fibonacci hashing can be implemented very efficiently by utilizing bit-shifting in which this power is used.

Table 1. Memory footprint of different sparse network implementations, measured in machine words per edge and per node. See the text for details and assumptions.

Implementation	Node	Edge
Linear probing	5	$2/\alpha$ ^a
Knuth [18]	8 ^b	5 ^c
IGraph [19]	1	2 ^d
Briggs [20]	2 ^e	$10 + 1/\alpha$ ^{b f g}
Packed array	4	2 ^d
Chaining STL hash table [17]	10	$5 + 1/\alpha$ ^{g h}

^aIf, for example, the maximum fill rate were 0.8 and the degree distribution even, the average fill rate α would be 0.6.

^bFor fairness of comparison, all node indices have been converted to machine words.

^cDue to implementation details, allocation overhead is not included.

^dIn practice, not all sizes of memory chunks are available for allocation, and thus some waste of space (corresponding to a fill rate less than unity) is inevitable. This does not apply to the linear probing hash table, as its size is constrained to powers of two, which are usually available.

^eUsed for node's degree and degree distribution

^fOne machine word has been added for otherwise absent weight.

^gAverage fill rate α can be reasonably approximated as unity.

^hThe minimum table size is 53 machine words.

In practice, we have utilized a network implementation based on the linear probing hash table in the study of very large social networks [4, 5]. The results were very satisfactory so that the primary test platform used for this Article was utilized for all the simulations in [4, 5] with no cost on the memory footprint or performance.

The results of the benchmarks clearly show that some data structures optimized for high cache miss latencies perform considerably better than conventional ones on modern hardware. Of course, the extremely sparse nature of complex networks makes their cache miss rates rather pathological. However, neither significant decrease in the latency of DRAM cells used in main memory nor any substitute technology for them are in sight. The problem of cache miss latency is therefore going to affect many other forms of computation, as the raw speed of processors is still expected to grow. Luckily, some data structures and algorithms developed in the past for external usage will probably be rather directly applicable as solutions. For example, database management systems commonly use trees with a large number of children per node, in order to make a node to fit as exactly as possible on a single memory page. In an analogous manner, nodes fitting exactly on processor cache lines would probably make memory-resident tree structures much more efficient.

At least one modern processor architecture, the Cell, tackles with the problem by introducing fast additional memory areas associated with each of its Synergistic Processing Engines (SPE) under a direct control of the programmer. Such an area serves as a kind of additional, manually controlled cache. Of course, the additional complexity of the structure quite necessarily makes programming harder. Algorithms developed for processing of data that can only be partially loaded to main memory with the rest remaining on mass storage, should be rather easily utilized on this kind of hardware.

Interestingly, in [22] a cache-aware hashing algorithm is introduced and tested on

the SPE, in addition to the more conventional Pentium 4. Although not suitable for our application due to rather large memory overhead, the solution has some very interesting features. In particular, *vector instructions* that have become available on last microprocessor generations have been utilized for key comparison. Several keys can thus be probed in parallel, providing performance enhancements both as such and by eliminating conditional branches in the code.

We are currently working on a complete software library on network analysis (to be published) based on the data structures described in this article. The library will be in the form of an extension to a scripting language, the basic structures and algorithms implemented in a low-level language for performance. This combination is expected to be able to combine high performance with easy usage and the possibility to perform rapid experimentation on networks.

References

- [1] Dorogovtsev, S. and Mendes, J., 2002, Evolution of Networks. *Advances in Physics*, **51**, 1079–1187.
- [2] Newman, M., 2003, The Structure and Function of Complex Networks. *SIAM Review*, **45**, 167–256.
- [3] Boccaletti, S., Latora, V., Moreno, Y., Chavez, M. and Hwang, D.U., 2006, Complex networks: Structure and dynamics. *Physics Reports*, **424**, 175–308.
- [4] Onnela, J.P., Saramäki, J., Hyvönen, J., Szabó, G., Lazer, D., Kaski, K., Kertész, J. and Barabási, A.L., 2007, Structure and tie strengths in mobile communication networks. *Proceedings of the National Academy of Sciences (USA)*, **104**, 7332.
- [5] Onnela, J.P., Saramäki, J., Hyvönen, J., Szabó, G., de Menezes, M.A., Kaski, K., Barabási, A.L. and Kertész, J., 2007, Analysis of a large-scale weighted network of one-to-one human communication. *New J. Phys.*, in press, preprint [arxiv:physics/0702158](http://arxiv.org/abs/physics/0702158).
- [6] Cantin, J. and Hill, M., 2001, Cache performance for selected SPEC CPU2000 benchmarks. *ACM SIGARCH Computer Architecture News*, **29**, 13–18.
- [7] Knuth, D.E., 1998 *The Art of Computer Programming*, 2nd Edition , Vol. 3: Sorting and Searching (Addison Wesley).
- [8] Lea, D., 2000, A Memory Allocator. Available online at: <http://gee.cs.oswego.edu/dl/html/malloc.html> (accessed 30. May 2007).
- [9] Tarjan, R.E., 1985, Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, **6**, 306–318.
- [10] Black, J.R.J., Martel, C.U. and Qi, H., 1998, Graph and Hashing Algorithms for Modern Architectures: Design and Performance. In: *Proceedings of the Proceedings of WAE*, August.
- [11] Janson, S., 2005, Individual Displacements for Linear Probing Hashing with Different Insertion Policies. *ACM Transactions on Algorithms*, **1**, 177–213.
- [12] Knuth, D. and Amble, O., 1974, Ordered hash tables. *The Computer Journal*, **17**, 135–142.
- [13] Celis, P., Larson, P.A. and Munro, J., 1985, Robin Hood hashing. In: *Proceedings of the Proceedings of the 26th IEEE Symposium on the Foundations of Computer Science* (IEEE Computer Society Press, Los Alamitos, CA), pp. 281–288.
- [14] Viola, A., 2005, Exact Distribution of Individual Displacements in Linear Probing Hashing. *ACM Transactions on Algorithms*, **1**, 214–242.
- [15] Patterson, D. and Hennessy, J., 2004 *Computer Organization and Design: The Hardware/Software Interface*, 3rd Edition (Morgan Kaufman).
- [16] Bayer, R., 1972, Symmetric Binary B-Trees: Data Structures and Maintenance Algorithms. *Acta Informatic.*, **1**, 290–306.
- [17] Musser, D.R., Derge, G.J. and Saini, A., 2001 *STL Tutorial and Reference Guide* (Addison Wesley).
- [18] Knuth, D.E., 1993 *The Stanford GraphBase: A Platform for Combinatorial Computing* (Addison Wesley).
- [19] Csardi, G., 2007, IGraph. Available online at: <http://cneurocvr.rmki.kfki.hu/igraph/> (accessed 12. Sep 2007).
- [20] Briggs, K., 2005, graphlib. Available online at: <http://keithbriggs.info/graphlib.html> (accessed 12. Sep 2007).
- [21] GNOME, 2007, GLib. Available online at: <http://library.gnome.org/devel/glib/stable/> (accessed 12. Sep 2007).
- [22] Ross, K., 2007, Efficient Hash Probes on Modern Processors. In: *Proceedings of the 2007 DBRank Workshop*, April.