T-106.290 Laboratory Course in Programming: The Significance of Data Representation to Performance

Assistent Ville Nenonen

Miro Lahdenmäki
55089K
mlahdenm@cc.hut.fi

Tuukka Lehtonen
51091A
tvlehton@cc.hut.fi

18th May 2004

Abstract

In this paper we studied the performance of depth-first search (DFS) with four different graph implementations. To our surprise in the tests our bit-matrix implementation didn't perform better than the integer-matrix implementation even though it needs 32 times less memory.

Our adjacency list implementation where we placed the data as it is common in graph implementations performed well in all cases, even when the graph data didn't fit to the cache. With the other list implementation there was some degradation in performance when the graph size exceeded a multiple of the L2 cache size.

When testing the influence of graph density the performance curve of the adjacency matrix implementations drew a parabolic curve indicating that performance was best with sparse and dense graphs.

1 Introduction

In this paper we study the significance of data representation to performance. We study this by comparing the efficiency of a basic graph algorithm called depth-first search on four different data implementations for a directed graph.

Caches have an integral effect on how efficiently we can utilize the power of modern processors. This has been a growing trend for many years and is likely to become more important as processor speed surpasses memory speed more and more. Modern machine architectures and caches are efficient when data can be handled locally so that is one of our primary concerns.

The reason why we focus on a basic graph algorithm is that within the limits of this course we cannot delve into a very extensive study.

1.1 Related Work

Dirk Grunwald [4] et al. pointed out in 1993 that programmers generally don't put too much thought on memory allocators and assume that the memory allocators provided by their programming environment are optimal. They demonstrate that poor reference locality reduces program performance by increasing paging and cache miss rates. Cache misses are becoming more crucial as the performance of memory relative to processor speed is decreasing all the time. Grunwald et al. show that space-efficient algorithms have poor reference locality often hindering performance. They suggest a memory allocator design that is fast and has good locality of reference.

In a previous paper they show that memory allocators customized for specific applications outperform general allocators distributed with widely-used operating systems while being more space efficient.

In this paper they find out that algorithms that search for free space for every allocation, such as FirstFit and GNU G++, are generally slower and have poor reference locality. Also an allocator that has been especially designed for good cache locality, Gnu Local, doesn't have significantly lower cache miss rates than BSD or QuickFit algorithms. These two algorithms allow very rapid allocation and deallocation and at the same time promote rapid object re-use thus leading to higher reference locality.

Black [1] et al. show that array-based lists are much faster than linked list implementations for sequential access. They accentuate that it is important to understand which variables can affect results. They point out that many papers on algorithms concentrate on higher level implementations and fail to take note of the cache characteristics of the machines used. Their studies show that for dense graphs an adjacency matrix using a bit-vector is the universal winner, while for sparse graphs an array-based adjacency list is best. They suggest that the best data structure depends largely on graph size and average node degree but that it doesn't depend on graph topology.

Chilimbi [2] et al. elevate that there are three general data placement designs that can be used to produce cache-conscious data structures. They are clustering, coloring and compression.

> Clustering attempts to pack data structure elements likely to be accessed contemporaneously into a cache block. - - Coloring segragates heavily and infrequently accessed elements in non-conflicting cache regions. - - Compression reduces structure size or separates the active portion of structure elements.

Chilimbi presents a cache-conscious memory allocator CCMALLOC that attempts to co-locate contemporaneously accessed data elements in the same cache block. It performs local clustering quite efficiently and is safe in that it affects only program performance. CCMALLOC differs from malloc in that it takes an additional parameter that points to an existing data structure element likely to be accessed contemporaneously with the element to be allocated. It is also quite easily utilizable.

Chilimbi also presents a complementary approach to cache-conscious allocation, to reorganize a structure's memory layout to correspond to its access pattern. For this he presents a cache-conscious tree reorganizer CCMORPH that applies clustering and coloring techniques. Tseng [7] extends CCMORPH to cluster acyclic graphs (DAGs) as well as trees. But we are experimenting with graphs that can be cyclic.

Tseng [7] also addresses the issue of cache performance with regard to data locality. In addition he stresses the need for both compile-time and run-time data locality optimizations. Both of these optimizations are of crucial importance when attempting to make high performance programming available to non-expert programmers. Although experiments have shown compile-time optimizations to improve performance, sometimes even dramatically, he presents three cases where compile-time optimizations are insufficient and gives optimization techniques for each of these cases.

Two basic representations of directed graphs are used in our experiments: adjacency-list and adjacency-matrix. Both of these can be implemented in several ways with regard to the data layout. In addition to these [3] presents the elementary depth-first search (DFS) algorithm which is under exprimentation in this paper. Cache-conscious allocation is a technique of particular interest in this field of experiments since it addresses caching problems in pointer based data structures, such as adjacency-list based graph representations.

In the following chapters we will discuss our experiment design and present the results and analysis of our experimentations. Experiment design covers the tested algorithm, discusses our input data considerations, testing factors and test run descriptions. The design is concluded with rough descriptions of the results to be shown. Experimentations are presented along with the resulting graphs and numeric data.

2 Experiment Design

2.1 Brief description of the algorithm

We use the basic depth-first search algorithm as it is described in Introduction to Algorithms [3].

As implied by its name, depth-first search seeks deeper in the graph whenever possible. Edges are explored from the most recently discovered vertex that still has unexplored edges leaving from it. When all the edges leaving from it have been explored, the search backtracks to explore edges leaving from the vertex from which it was discovered. This process continues until all the vertices that are reachable from the original source vertex are discovered. If there are still undiscovered vertices, then one of them is selected as a new source. Depth-first search is ready when all the vertices are discovered.

Vertices are colored during the search to indicate their state. Each vertex is initially white. When a vertex is **discovered** it is grayed and when its adjacency list has been examined completely it is blackened. This guarantees that each vertex ends up in just one depth-first tree.

Each vertex is also **timestamped** twice, when the vertex is first discovered and when the search finishes on the vertex. The timestamps are integers between 1 and $2|V|$, where $|V|$ is the number of vertices.

2.2 Brief description of the technology

The two standard ways to represent a graph are as a collection of adjacency lists or as an adjacency matrix.

> The **adjacency-list representation** of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each $u \in V$, the adjacency list $Adj[u]$ contains (pointers to) all the vertices v such that there is an edge $(u, v) \in E$.
>
> For the **adjacency-matrix representation** of a graph $G = (V, E)$, we assume that vertices are numbered $1, 2, \ldots, |V|$ in

some arbitrary manner. The adjacency-matrix representation of a graph G then consists of a $|V|$ x $|V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

2.3 Description of input data

We will create the input data with Knuth's Stanford GraphBase [5] (SGB). With it we can easily generate random graphs with few parameters. We will test different sized graphs with varying density.

Black et al. [1] point out that the adjacency-matrix representation is most efficient for dense graphs where as an array based adjacency-list representation is more efficient for sparse graphs. They observed no particular effect by varying the graph topology. We will perform some tests to see if this is true for our case also.

We will place the graphs generated with SGB to our own simplified data structures.

The generated graph data will be tested on two different general data representations: an array based adjacency-list representation and an adjacency-matrix representation. The matrix representations will be referred to as *adjmat* and *adjbitmat*, for *adjacency integer-matrix* and *adjacency bit-matrix* representations. In the adjacency-list representation we will test two different data placement schemes. In the first scheme (*adjlist1*) we treat a vertex and its outgoing edges (vertex pointers) as a variable sized successive memory block. Such blocks are placed consecutively in a single allocated block of memory big enough for the whole graph. In the second scheme (*adjlist2*) we will use a different data layout and separate the array of graph vertices from the edge lists of each vertex. Thus each vertex will be constant sized and contain a pointer to its list of edges. Figures 1 and 2 clarify these memory layouts. Although somewhat synthetic, *adjlist2* is closer to the common adjacency-list representation than *adjlist1*.

2.4 List of parameters

SGB gives us the possibility to vary the following parameters in graph generation:

- Vertices
- Edges
- Multi

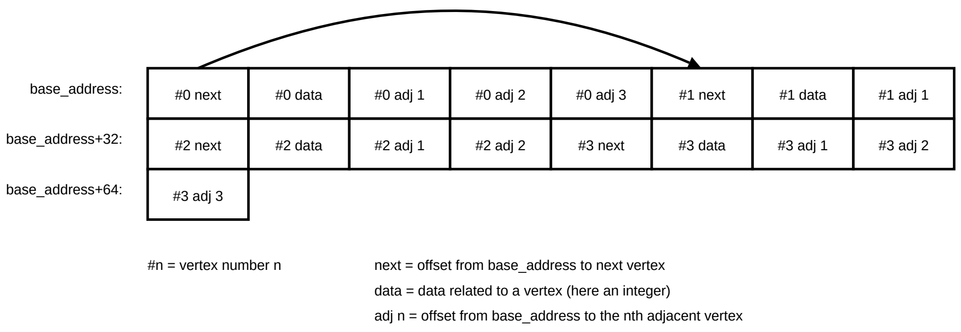

Figure 1: *adjlist1* memory layout

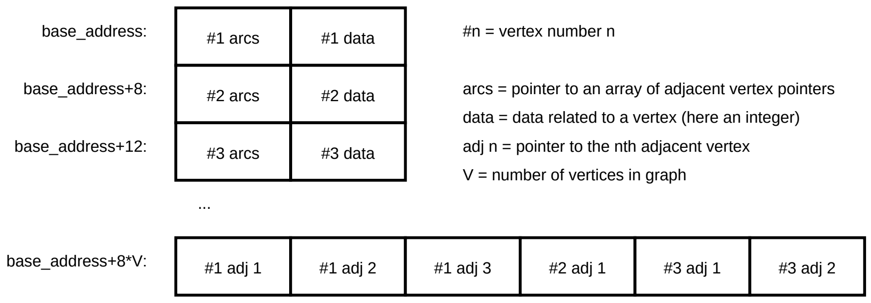
base_address:
base_address+8:
base_address+12:
base_address+8*V:
#1 arcs
#1 data
#2 arcs
#2 data
#3 arcs
#3 data
...
#1 adj 1
#1 adj 2
#1 adj 3
#2 adj 1
#3 adj 1
#3 adj 2
#n = vertex number n
arcs = pointer to an array of adjacent vertex pointers
data = data related to a vertex (here an integer)
adj n = pointer to the nth adjacent vertex
V = number of vertices in graph

Figure 2: *adjlist2* memory layout

- Self
- Directed
- Distance from
- Distance to
- Min length
- Max length
- Seed

From these parameters we choose only vertices and edges as factors to keep the amount of testing manageable. Multiple edges between two vertices or edges leading back to the same vertex don't make any difference in DFS so we don't allow them. Also we are studying directed weightless graphs so we don't need any length parameters. We will also keep the seed constant.

Vertices and edges define the *size* and *density* of the graph. We will study the efficiency effects of size and density separately on our four test cases.

We will vary the number of vertices from 30 to 3500 and the edge density from 5% to 95%.

With the distance from and distance to parameters we can affect the probability of incoming and outgoing edges at each vertex and thus cause clustering in the graph. Without this parameter a uniform graph will be given. We chose not to vary these parameters and use the uniform edge distribution.

With the seed parameter provided by SGB we can control the random seed by which the graph is generated in a system independent fashion. So by giving the same set of parameters we can generate the same graph on different platforms.

2.5 Environments

We performed our tests on 2.4 GHz Intel Pentium 4 machines with 533 and 800 MHz FSB. The P4 machines have 8 KB L1 data cache and 512 KB L2 cache. Both run Linux as their operating system.

We used the GNU C/C++ compiler suite for compiling our test programs. All tests were compiled with full optimizations.

2.6 Description of the test runs

We found out that SGB can be very slow in generating especially dense graphs and can take thousands of times longer than executing a DFS sample run on the finished graph itself. So we decided to optimize the procedure. First we create the graphs with SGB for the test cases with predefined parameters and save the graphs to disk. The graphs are then converted into all of our graph representations and they are in turn saved to the disk for later use.

Time is measured with the `clock()`-function provided by the operating system which measures process time. The time for one depth first search can be so small that it's impossible to measure it accurately using `clock()`. For this reason we have to run DFS several times for one sample. We control this with the -r parameter which stands for *runs-per-sample*. There can also be a difference of 4 orders of magnitude in the running time of the samples in our planned graphs. For this reason we have to change the runs-per-sample value even among samples that go to the same graph. We tried to keep the time it takes to run one sample in the scale of seconds but less than 10 seconds to keep the time required to run all the tests humane. As the runs-per-sample value varies inside a graph we have to scale the result times according to the runs-per-sample value.

In our tests we vary the graph size (number of vertices) and its density (edges per vertices squared). We run each test sample fifty times and control with the runs-per-sample that each sample takes more than a second. As the `clock()`-function will overflow approximately every 72 minutes we check with each sample if that has happened and run the sample again if it has. After a sample run we discard the obviously erroneous values based on how much they differ from the median of the samples. These samples are rerun. This may happen when some other program interferes too much causing extra cache misses. We plot the average value of the samples in a graph and write down several key figures including the mean, variance, standard error and confidence interval.

We study the influence of graph size using ten different sizes ranging from 30 to 3500 vertices on three different densities 5%, 40% and 75%. We study the effect of density using ten different values from 5% to 95% on three different graph sizes 50, 400 and 750. The slowness of generating dense graphs with SGB is one of the factors that limits our graph sizes.

2.7 Results

We use three graphs to show the effect of graph size, one for each tested density. Each graph has 10 result values per implementation.

Similarly we use three graphs to show the effect of graph density, one for each tested size. Each graph has 10 result values per implementation.

We take 50 samples of each factor combination. The statistical significance of the results is ensured by testing that the confidence interval of each result falls within $\pm 2.5\%$ of the mean value. Samples that are clearly erroneous are discarded automatically during testing. Samples that differ more than 20 % from the sample median at any time during a test run are discarded since they have been disturbed by other processes.

3 Experiments

3.1 Varying Graph Size

In this experiment we experimented how graph size affects the performance of DFS. We did the experiments with three graph densities (d) 5%, 40% and 75%. The number of vertices was linearly raised from 50 to 3500 vertices in the first figure and from 50 to 1000 in the last two figures. The number of edges can be counted from equation (1).

$$|E| = d|V|^2 \tag{1}$$

The results are shown in figures 3, 4 and 5 with each having a constant density while increasing the number of vertices.

The computational complexity of the DFS algorithm follows from the graph implementation. Adjacency matrix implementations have $|V|^2$ complexity and adjacency list implementations have $|V||E|$ complexity.

After this we will refer to the different tested implementations as follows:

- adjmat = adjacency integer-matrix
- adjbitmat = adjacency bit-matrix
- adjlist1 = adjacency list implementation 1
- adjlist2 = adjacency list implementation 2

In Figure 1 *adjlist2* performs best and its performance is almost linear. *Adjlist1* performance follows closely with *adjlist2* performance but starts to lag behind on 1500 vertices and from just over 3000 vertices on its performance lags behind some more when compared to *adjlist2*. With 3500 vertices *adjlist1*'s performance is about a third from *adjlist2*.

The performance of *adjmat* and *adjbitmat* lags behind much quicker than *adjlist1* when compered to the *adjlist2* performance. Their performance curve resembles a parabola. *Adjbitmat* performs a little better than *adjmat*. With 2700 vertices their performance is about a tenth of *adjlist1* performance as well as with 3500 vertices.

In Figure 2 *adjlist2* performs best but takes a slight notch at about 500 vertices. *Adjlist1* performance starts to lag behind at about 350 vertices and ends up at 2/3 performance with 1000 vertices. The performance of *adjmat* and *adjbitmat* are very close to each other and are left behind in performance ending up in 1/4 of the performance of *adjlist2*. *Adjmat* wins the race with *adjbitmat* by a very small margin.

In Figure 3 *adjlist2* performs best once again with near linear performance. It takes a notch at about 350 vertices. *Adjlist1* starts to leave behind at 250 vertices and takes a hit at 600 vertices. It ends up last in this race with 1/3 of *adjlist2* performance.

Adjmat and *adjbitmat* perform similarly following a parabola. They overcome *adjlist1* at 500 vertices and end up with 2/5 of *adjlist2* performance.

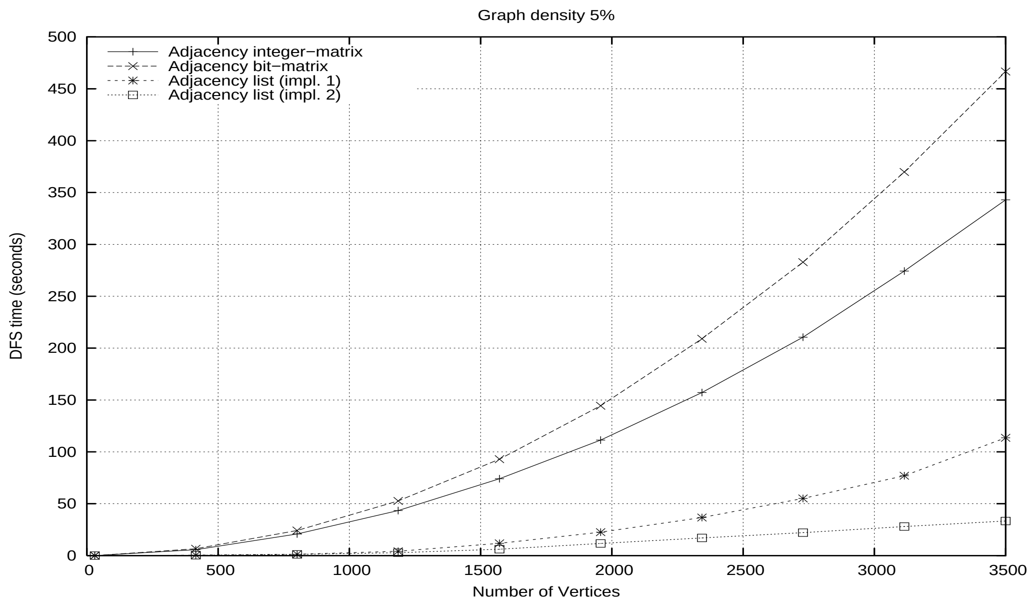

Figure 3: Size test - 5% density

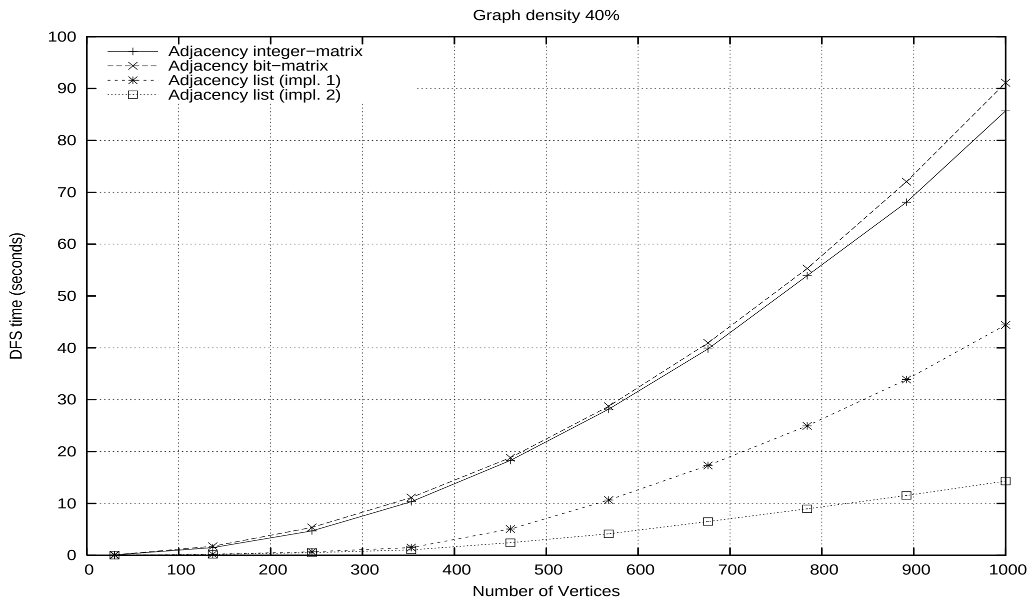

Figure 4: Size test - 40% density

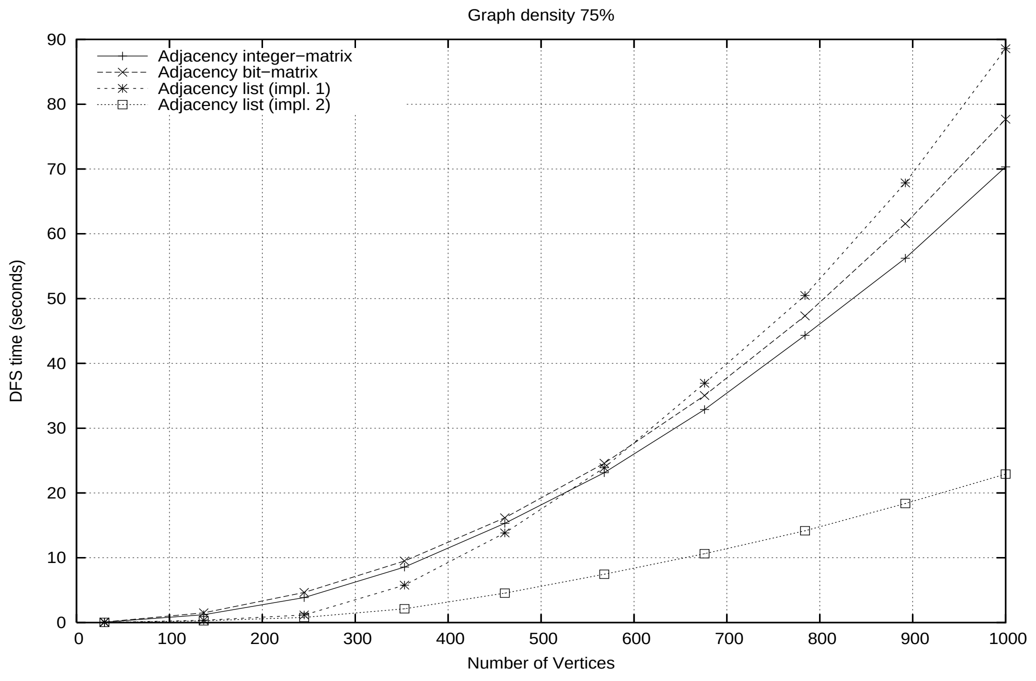

Figure 5: Size test - 75% density

3.2 Varying Graph Density

In this experiment the purpose was to see how graph density, i.e. how many edges are there per vertex, affects performance on different graph layouts and to compare the different layouts agains each other in a meaningful way.

Factor combinations:

Number of vertices: $|V| = 50, 400, 750$

Density: $d = 0.05 + 0.1N$, where $0 \leq N \leq 9$.

The number of edges can be calculated by equation (1).

The computational complexity is the same as in section 3.1.

The results are shown in figures 6, 7 and 8 with each having a constant number of vertices with increasing density.

All adjacency matrix traversals seem to follow the same pattern of a downward opening parabola where the worst performance appears to be at approximately $d = 0.5$. Both *adjmat* and *adjbitmat* follow the same pattern but *adjbitmat* constantly lags a little behind *adjmat*.

On the other hand, adjacency list traversals show different behaviour. Just as in the tests in section 3.1, *adjlist1* starts showing bad performance in comparison to *adjlist2* as we increase |V|. It even starts losing to the matrix implementations at $d = 0.8$ when $|V| = 400$ and at $d = 0.7$ when $|V| = 750$.

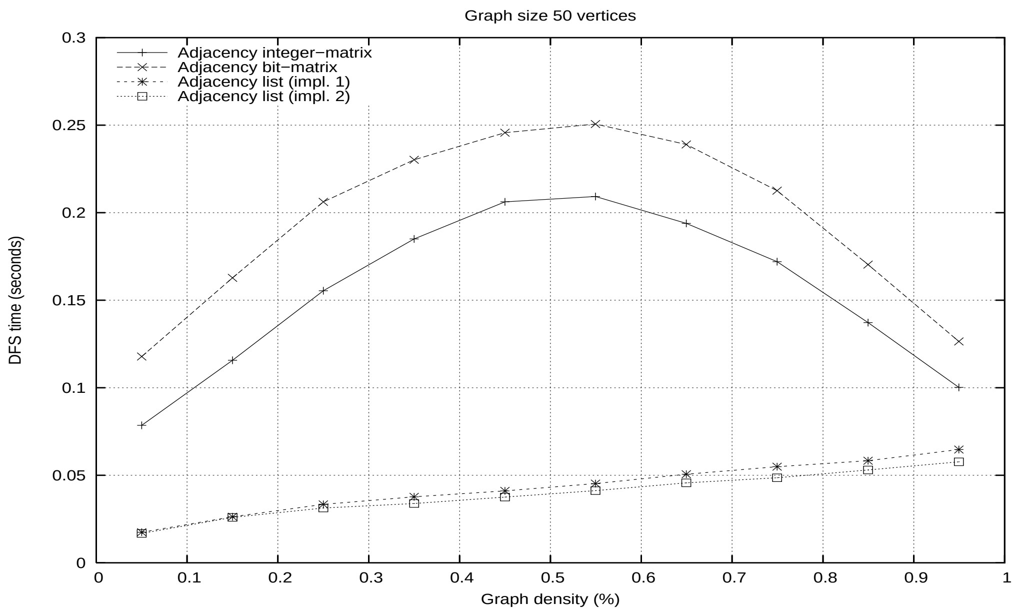

Figure 6: Density test - 50 vertices

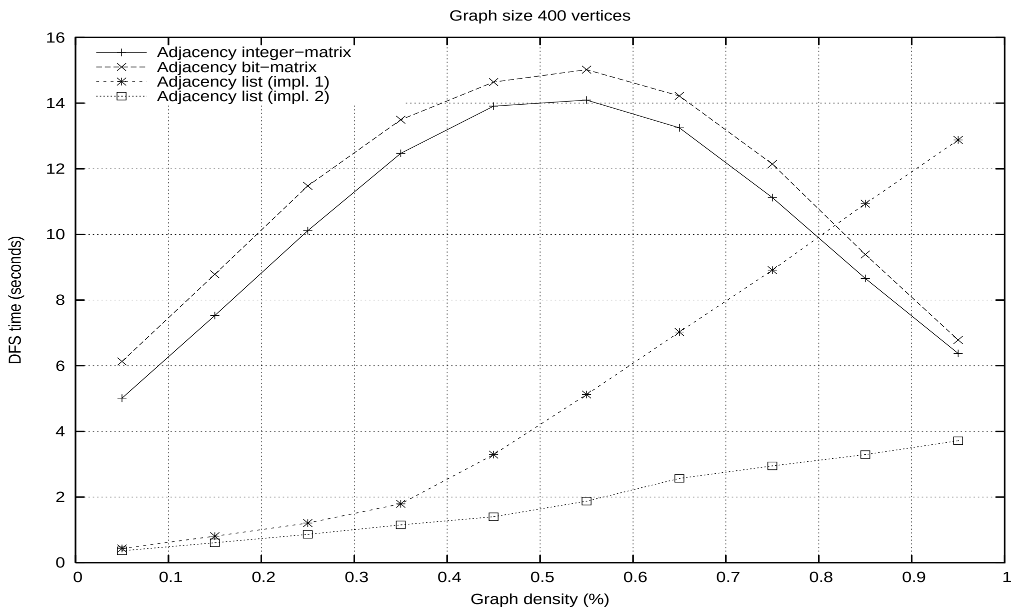

Figure 7: Density test - 400 vertices

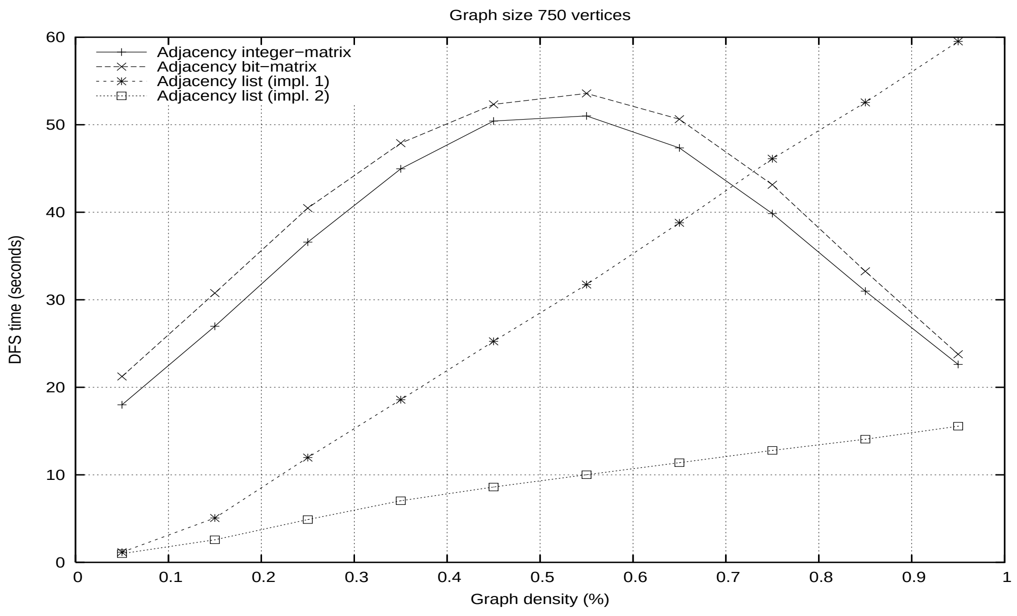

Figure 8: Density test - 750 vertices

4 Analysis

4.1 Varying Graph Size

In general, there were no huge surprises here although there are some puzzling results.

First of all the performance of *adjmat* doesn't shock in any way. It pretty much follows the parabolic trend it should be following based on it's complexity in each of the three graph density cases. On the other hand, the performance of *adjbitmat* was both a disappointment and a surprise based on the results by Black et al. [1] although they tested *breadth-first-search* (BFS) instead of DFS. Our non-optimized implementation may affect this, but another point of view is that, since we do not perform any "payload operations" during traversal that we probably in real life would, the resulting inner loop is a tight one. Therefore it would seem that the bit level calculations done in the *adjbitmap* are just enough to make the code generally slower than the *adjmat* case. Even huge differences in the amount of memory needed on large graph sizes (10000 vertices $\Rightarrow$ 382 MB vs. 12 MB) don't seem to turn the tables as one might expect. One thing to note here is that in most cases DFS jumps around a lot in memory, and doesn't process much successive memory in a row. This raises the probability that earlier loaded cache lines get trashed while traversing the subgraph.

In figure 1 we can see how *adjlist1* takes a slight performance hit when the graph no longer fits in L2 cache (1500 vertices) and another when the graph is twice the L2 cache size (3500 vertices).

In figure 2 we can see *adjlist1* and *adjlist2* take a slight performance hit when the graph is greater than half the L2 cache size (450 vertices).

In figure 3 we can see *adjlist1* take a slight performance hit when the graph is greater than half the L2 cache size (350 vertices). It takes another when the graph is greater that L2 cache size (450 vertices) and continues to perform worse and worse.

4.2 Varying Graph Density

All of the graph density varying tests show the same common behaviour in the matrix-based tests. Both with very sparse and very dense graphs performance is much higher than with medium density graphs. This would seem to be caused by the general patterns of traversal in the adjacency matrix. To clarify, take the two extreme examples: a fully disconnected graph (0% density) and a fully connected graph (100% density). Considering how the DFS algorithm traverses these adjacency matrices in memory, it can be seen that in the fully disconnected case, the matrix is just scanned through linearly thus exploiting maximum data locality and avoiding bad cache effects. In the fully connected case the graph vertices are visited in linear order, hence memory is again scanned in a very linear fashion. On the other hand, in the medium density cases, traversal is likely to be more random in memory and therefore likely to take more time. This reasoning is very much supported by figures 6, 7 and 8. Just as in size testing, *adjbitmat* constantly shows worse performance than *adjmat*.

The *adjlist1* and *adjlist2* cases show that the two different layouts perform almost equally up until the graph size draws near to and exceeds the L2 cache size. After that *adjlist1* execution time starts growing 3–4 times as fast as *adjlist2*. We used *Valgrind* [6] and its plugin *Cachegrind* to see how the cache hiearchy reacts to the adjacency-list tests. Simple Valgrind runs suggested that the *adjlist1* case causes 10-30% more L1 data cache misses than *adjlist2*.

5 Summary

What we did was test four different graph implementations to experiment their effect on DFS efficiency. We had two adjacency matrix implementations and two adjacency list implementations. To our surprise our adjacency bit-matrix implementation didn't perform better than the integer-matrix implementation even though it fit in a fraction of the memory needed by the integer-matrix implementation.

We experimented with a common adjacency list implementation *adjlist2* where vertices and edges are placed apart. It performed very well on all test cases and demonstrates how much data presentation and data placement can influence efficiency.

Our other adjacency list implementation *adjlist1* in which vertices and edges are alongside suffered somewhat when the graph data didn't fit in L2 cache and on bigger and denser graphs it was left behind by the adjacency matrix implementations.

When we tested the influence of graph density the performance of the adjacency matrix implementations was best with sparse and again with dense graphs.

What left us curious was the poor performance of the adjacency bit-matrix and for future research it would be interesting to try to make it perform better. Another point of interest would be to test the implementations with different payloads. Also it might be interesting to draw the graphs using the memory needed for the graph as the x-axis.

References

[1] J. Black, C. Martel, and H. Qi. Graph and hashing algorithms for modern architectures: design and performance, 1998. `http://citeseer.nj.nec.com/black98graph.html`.

[2] T. M. Chilimbi, M. D. Hill, and J. R. Larus. Making pointer-based data structures cache conscious. *Computer*, 33(12):67–75, 2000. `http://citeseer.nj.nec.com/chilimbi00making.html`.

[3] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[4] D. Grunwald, B. G. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–186, 1993. `http://citeseer.nj.nec.com/grunwald93improving.html`.

[5] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing.* 1993. [From the publisher]: ... represents Knuth's final preparation for Volume 4 of *The Art of Computer Programming.* Through the use of about 30 examples, the book demonstrates the art of literate programming. Each example is a programmatic essay, a short story that can be read by human beings, as well as read and interpreted by machines. In these essays/programs, Knuth makes new contributions to the exposition of several important algorithms and data structures.

[6] J. Seward. Valgrind, a gpl'd system for debugging and profiling x86-linux programs, 2002-2004. `http://valgrind.kde.org`.

[7] C.-W. Tseng. Software support for improving locality in advanced scientific codes, 2000. `http://citeseer.nj.nec.com/article/tseng00software.html`.