Scalability for High Cardinality in Steerable MDS - Final Report

Allan Rempel*

University of British Columbia

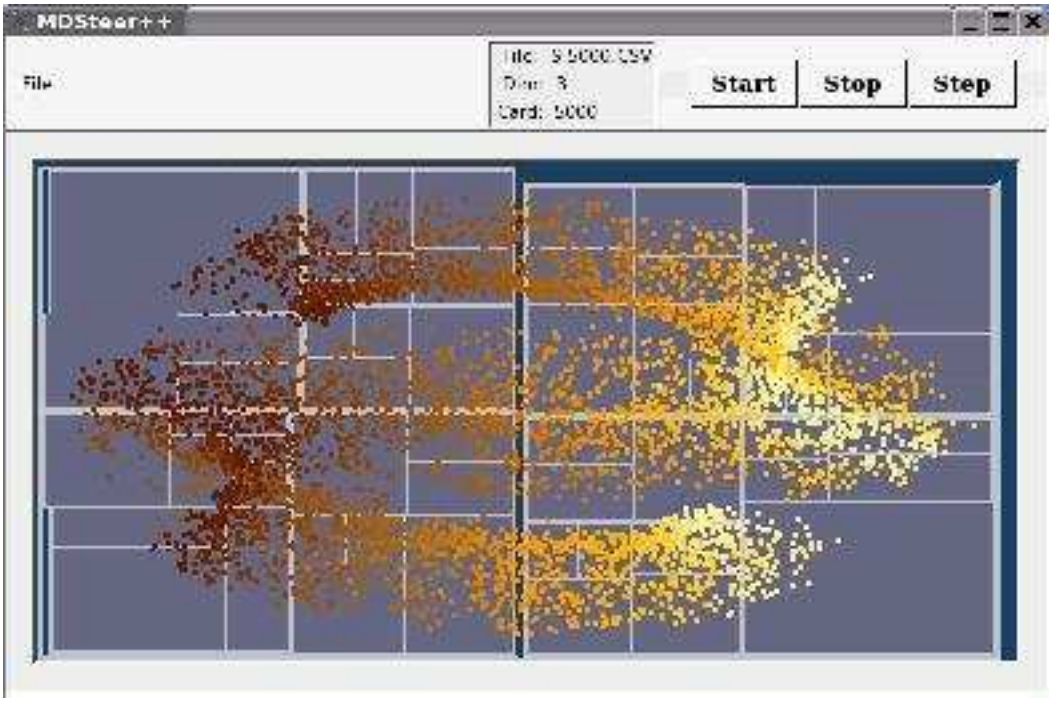

Figure 1: MDSteer++ in action

ABSTRACT

Multidimensional Scaling (MDS) is a dimensionality reduction technique that is concerned with preserving the high-dimensional distance between points when those points are mapped to a low-dimensional (typically 2-D) space. A number of techniques have been developed in which the time complexity has been reduced from $O(N^3)$ to $O(N^2)$[4] to $O(N\sqrt{N})$[1][2] to $O(N\sqrt[4]{N})$[7] to $O(N \log N)$[6]. In the process, however, the space complexity has increased to $O(N^2)$ in that a matrix of precomputed distance values is needed so that these values need not be repeatedly recalculated. This matrix limits the size of the data set that can be processed according to the main memory of the computer.

I demonstrate the use of techniques for offloading this data, using a file system and a database system, and discuss the effects and how they demonstrate issues of scale in MDS. These techniques are used in MDSteer++, a software system that performs progressive user-steerable MDS. I describe the code development necessary to achieve these goals, and outline areas of further research involving MDS and MDSteer++ in particular. I also outline further code development that would be appropriate in order to be able to release MDSteer++ as a robust modular software system that could easily be integrated into the code development efforts of others interested in MDS.

*e-mail: agr@cs.ubc.ca

CR Categories: K.6.1 [Management of Computing and Information Systems]: Project and People Management—Life Cycle; K.7.m [The Computing Profession]: Miscellaneous—Ethics

Keywords: MDS, multidimensional scaling, database

1 INTRODUCTION

Theory and practice must meet. Either can exist without the other, and they sometimes do, but where theory and practice do not meet, the result is often unsatisfactory. *Praxis*, the process by which abstract concepts are connected with experienced reality, should be a consideration wherever computer computer software is written.

This project is concerned with some practical considerations associated with multidimensional scaling. One of the most significant practical considerations of any software system is scalability.

1.1 Scalability

Scalability is an important consideration in the development of many software systems. It is not hard to write software that works well on small data sets, but the real test of a system's mettle is how well it works on large data sets. In my experience as a software

developer in the computer animation industry, scalability is not always an easy thing to develop into a software system. It's easy to write a piece of software that processes 1000 data items, but what happens when you have 1,000,000 data items? What happens when 600 users all want to work on the same project at the same time? What happens when file sizes grow beyond 4 GB? (4 GB is the maximum size representable in a 32-bit unsigned integer, which along with the 32-bit signed integer type (whose maximum value is half that) are the most common formats for representing file sizes.)

There are many facets of scalability. It is important to know both the theoretical (order-notation) and practical efficiency of a system. Even though constants may not be considered significant in theoretical analysis, when a system runs twice or ten times as fast as another, that is significant. It is also necessary to consider efficiency in space (memory, disk, etc.) as well as in time. One way to deal with time complexity issues is to use a progressive algorithm, as that can have a huge mitigating effect on the time cost the algorithm would otherwise have. In all these things, the effective use of system resources such as memory, disks, networks, etc. is important in avoiding wasted energy, which results from thrashing and other behaviour modes that indicate the computer is operating beyond its reasonable limits.

Even though more focus is traditionally placed on evaluating time complexity, space complexity can actually be more of a limiting factor. Time is more or less infinite; if an algorithm will run in a reasonable time for data of size N, it will still likely run in a reasonable time for data of size $N+1$. However, disk space, memory, bandwidth, and other system resources are generally finite, and with respect to those, it is possible to have a situation near the limits where an algorithm will run on data of size N but not on data of size $N+1$.

While this project focuses on scalability with respect to the cardinality of the set of data points in MDS, dimensionality is also something that should be considered, even though in practice, the number of dimensions is vastly less than the number of points. Nonetheless, incrementing the number of dimensions can have a much more deleterious effect on performance than incrementing the point count, which makes it worth consideration.

1.2 Multidimensional Scaling (MDS)

Multi-dimensional (also called high-dimensional or multi-variate) data sets can arise from any of a number of different sources. Any process that produces data that has a number of different variables can be said to be multi-dimensional, where each variable is its own dimension. Data from the social sciences, the financial sector, bioinformatics, and other sources often produce multi-dimensional data sets, and it is often desirable to be able to see patterns in the data. Unfortunately, we cannot generally comprehend data of more than 3 dimensions that is visually or geometrically laid out, because we live in a 3-dimensional universe.

Various techniques have been devised to deal with with this limitation by mapping the high-dimensional data into a 3-dimensional or 2-dimensional space, because we are better able to understand and process data represented that way. (Usually the focus is to map to a 2-D space, because computer monitors are 2-D displays and thus even a 3-D space still needs to get reduced to 2-D in order to be displayed.) These are generally called dimensionality reduction techniques. However, such mappings are necessarily limited in their ability to accurately represent high-dimensional data, in the same way that a line segment is a limited 1-D representation of a 2-D circle, which itself is a limited 2-D representation of a 3-D sphere. The nature of these limitations can vary between the different techniques, and thus different techniques could be seen to serve different goals.

Multidimensional Scaling (MDS) is a *dimensionality reduction* technique that produces a p-dimensional embedding of data in a q-dimensional space (where $p < q$). It specifically aims to preserve the distance relationships between points, so that points that are far apart in the high-dimensional space are also far apart in the low-dimensional (typically 2-D) space, and points that are close together in the high-dimensional space are also close together in the low-dimensional space. (Generally, a standard Euclidean distance metric has been used for this calculation, but other metrics could also conceivably be used.) However, the actual absolute placement of points in the low-dimensional space does not necessarily correspond to their placements in the high-dimensional space, or to anything else for that matter. In fact, the MDS techniques discussed in this paper all begin by randomly placing points in the 2-D space. A spring model is then employed to move the points to more appropriate locations (relative to the other points). The net effect of this technique is that clusters of points, the relationships between those clusters, and general large-scale features that stand out from the noise in the high-dimensional space will be preserved in the low-dimensional space; this is in fact the purpose of MDS.

It is worth noting that MDS is generally run on *abstract* data, which is data that does not have an inherent geometric placement in any space, high-dimensional or low-dimensional. For instance, financial data, where some dimensions may be different kinds of expenses, could be laid out somewhat arbitrarily in a high-dimensional geometric space because there is no inherent geometry to the numbers. This is in contrast to something like a set of 3-D points produced by a particle system in the context of a computer animation simulation, where the absolute geometric positions of the points are very important.

It is also worth noting that the MDS implementations discussed in this paper require their input to be *point data* as opposed to geometric shape data such as manifolds, polygonal or spline representations of geometric figures, etc.

Figure 2 shows an example of MDS running on a 2-D data set consisting of a 500-node small world network, 'scaling' it down to 2-D and producing another 2-D data set. Notice that the clusters and the relationships between the clusters are preserved, even though the absolute positions and orientations of the clusters are not. (If the source data set were of high dimension, it would be meaningless to speak of preserving the absolute positions of points.) This sort of transformation from 2-D to 2-D is an important sanity-check; if it works well, it gives confidence that transformations from higher-dimensional spaces which we may not be able to visualize would also work well.

2 Previous Work

Quite a lot has been written about MDS previously, and a number of algorithms have been proposed which perform the task with varying degrees of efficiency.

2.1 Time Complexity Matters

Initial algorithmic approaches to MDS used an eigenvector analysis of an NxN matrix, resulting in an $O(N^3)$ time complexity. They also would require recomputation of the whole algorithm if the data were to change even slightly. Subsequent algorithms used a force-based scheme, where points would initially be randomly placed in the 2-D space and a spring model would be used to pull together those points that are close in the high-dimensional space (in particular, closer than they are in the 2-D space) and push apart those points that are farther in the high-dimensional space than in the 2-D space. This results in approximate solutions rather than exact solutions, and the difference between these can be quantified by the *stress* remaining in the system when the algorithm is finished, which is the sum of the stresses in the individual springs. A single spring stress between 2 points can be seen as a measure of the

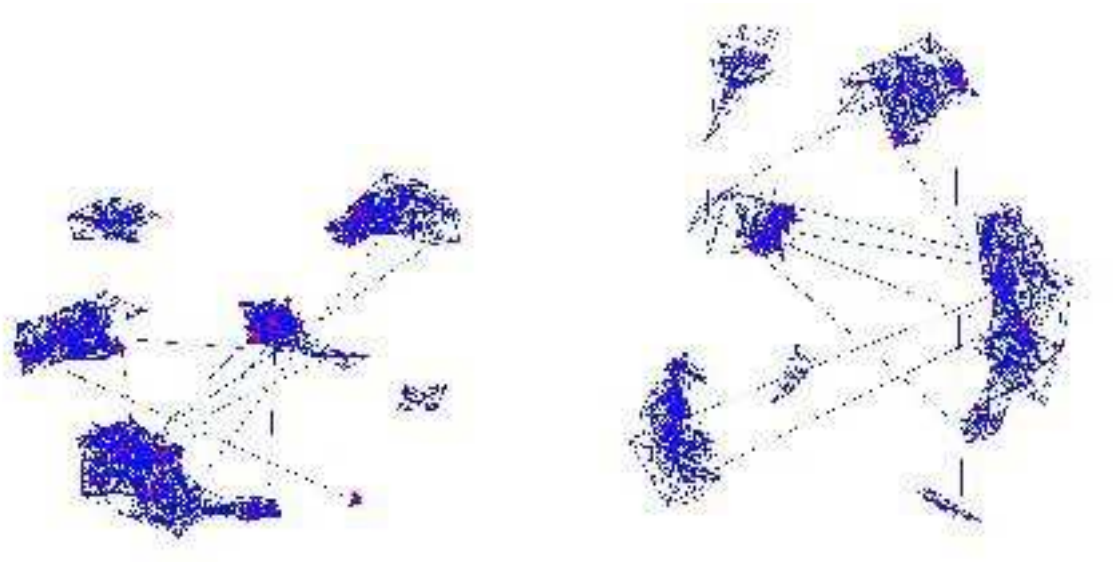

Figure 2: Example of MDS transformation of 2-D small world network data set

difference between the optimal (theoretically) and actual distances between those points. The approximate and iterative nature of this algorithm opens the door to creative new more-optimal algorithmic techniques as seen below. The basic implementation of the spring model MDS still results in $O(N^3)$ time complexity, because for each point we consider the spring forces of all other points in deciding how to adjust the point, and generally, N iterations of that are required to reach an appropriate level of convergence. The spring force acting on a point is given by

$$Stress = \frac{\sum_{i<j}(d_{ij} - g_{ij})^2}{\sum_{i<j} g_{ij}^2} \quad (1)$$

where d_{ij} is the high-dimensional distance between two points i and j and g_{ij} is the distance between those points in the low-dimensional space.

In 1996, Chalmers published an algorithm which reduced the time complexity to $O(N^2)$ by computing each point's spring forces against only a constant-size set of points, rather than all the other points. [4] This was followed in 2002 by an $O(N\sqrt{N})$ algorithm by Morrison, Ross, and Chalmers in which a subset of points of size $\sqrt{N}$ was run through the $O(N^2)$ algorithm resulting in a set of well-placed anchor or *parent* points (as they have since become called) with a time complexity of $O(N)$ for that phase. The second phase was to add and adjust all the remaining points by positioning them relative to the set of parents, which for N iterations results in a complexity of $O(N\sqrt{N})$.[1][2] Morrison and Chalmers then further refined their technique by using a $\sqrt{\sqrt{N}}$ size subset of their parent set for their spring calculations to bring the overall complexity down to $O(N^{5/4})$ or $O(N\sqrt[4]{N})$. [7]

Chalmers *et al* also ran experiments on a 3-D data set ranging in size from 5000 to 50,000 points and on a 13-D data set ranging in size from 2000 to 24,000 points, in which they confirmed that the more efficient $O(N\sqrt{N})$ algorithm ran substantially faster than (taking less than 1/3 the time of) the previous $O(N^2)$ algorithm. Interestingly, they also observed that the resultant placement of points was actually better with the less expensive algorithms than with the more expensive algorithm. The residual stress in the system was between 15% and 30% less with the $O(N\sqrt{N})$ algorithm than with the $O(N^2)$ algorithm. They also compared the performance against the full $O(N^3)$ spring model algorithm and in addition to the huge expected speed improvements (9 seconds vs. 577; 24 seconds vs. 3642), they also observed that the residual stress in the cubic time algorithm was over 3 times that of their $O(N\sqrt{N})$ algorithm.

2.2 Time Complexity Doesn't Matter

Jourdan and Melançon made one more improvement to this series of algorithms, bringing it to $O(N\log N)$. The algorithm achieves this by creating a constant size subset $P \subset S$ (where S is a size $\sqrt{N}$ subset of the whole set of points) and then $\forall p \in P$ creating a sorted list L_p consisting of points $s \in S$. Because these lists are sorted, a binary search can be performed on them for each point that needs to be placed, resulting in $O(N\log N)$ time complexity.[6]

Jourdan and Melançon also ran experiments similar to those of Chalmers *et al* and made some interesting observations. First, they noted that theoretically, for $N < 5500$, the $O(N^{5/4})$ algorithm actually performs better than their $O(N\log N)$ algorithm. Moreover, the $O(N\log N)$ algorithm doesn't really begin to show much of an improvement over the $O(N^{5/4})$ algorithm until N gets to around 75,000 points. Their experimental data also showed that there was only minimal difference in the performance of the $O(N^{5/4})$ and $O(N\log N)$ algorithms for N less than about 70,000.

They then made another improvement by introducing "Multi-scale MDS" in which they incrementally expanded the scale of dissimilarities (increasing the distance threshold). With this algorithm, which does not improve on the $O(N\log N)$ time complexity, they experimentally observed that it completed in less than half the time of the previous $O(N\log N)$ algorithm, and with a residual stress level that was between 0% and 75% less than that of the previous algorithm on data set sizes of up to 10,000 points.

2.3 Lots of Things Matter

Clearly, theoretical time complexity calculations and predictions tell only part of the story. The part they tell best is what happens when data sets become very large. This is of value of course because we want our algorithms to scale as well as possible in order to use them on large real-world data sets. However, there are are other considerations of scale besides time complexity, and one might ask how often one actually runs these algorithms on really large data

sets, or how capable the hardware on which the algorithms are being run is of scaling up to very large data sets.

Munzner *et al* have addressed these issues of scale by introducing *steerable* and *progressive* MDS.[11] The progressive aspect comes from immediately and continuously displaying the points in the 2-D space as their positions are calculated, thereby giving the viewer real-time information on the progress of the placement. The steerable aspect comes from introducing a level of interactivity by allowing the viewer to select areas of the screen on which to concentrate further processing. Together, these two aspects allow a viewer in an interactive environment to obtain the desired information from the MDS process in less time than with previous methods, because unnecessary avenues of computation are put on the back burner while more relevant avenues are processed more quickly than they would be otherwise. (The progressive aspect is of particular interest to me as my previous research included work on progressive transmission of images using wavelets.[9])

They further refined this technique by rewriting the MDS architecture in C++ (where it was previously implemented on the Java-based HIVE infrastructure of Ross and Chalmers[10]) using a GUI written with the Qt library and OpenGL. They also made the technique more fully progressive than it was before, and identified and addressed some artifacts of the steerability.[5] The result is MDSteer++, which is the software system that is the focus of this paper. The code base of MDSteer++ is portable between Windows and Linux, with only minor changes in the makefile. Figure 3 shows a screen shot of the program running on a 5000-point extruded S curve data set.

With the transition of the MDS algorithm from one that would run without user input (and could thus run in the background) to an interactive process, the actual and perceived speed of processing (apart from the time complexity) became increasingly important considerations. Anything that could noticeably improve that performance would be helpful.

One characteristic of MDSteer++ is that it precomputes the high-dimensional distances between all the points and stores them in a distance matrix whose size is given by

$$Size = \frac{points^2 - points}{2} \tag{2}$$

so that they need not be repeatedly recalculated. Thus, while the speed of the system has improved, the space complexity has increased to $O(N^2)$ This matrix limits the size of the data set that can be processed, based on the available main memory of the computer. Offloading this matrix to an area of subsidiary storage would significantly help improve the ability of any given computer to process a much larger data set.

Theoretically, MDSteer++ was designed to be able to handle data sets of 300+ dimensions and 1,000,000+ points. However, practically, it is of course limited by system resources, in particular, memory.

3 Reducing the Memory Footprint

The question which spawned this whole project is whether offloading the distance matrix to a database would yield an improved ability to run MDSteer++ on large data sets. In the process, it occurred to me that databases consist of software, file systems, and overhead, and that if MDSteer++ were to just write the file directly to the local disk, that would likely be faster (and perhaps a better test of offloading the distance matrix) than using a database. In the end, I did both.

The coding component of this research consisted primarily of finding ways to offload the distance matrix mentioned in Section 2.3 so as to help alleviate the limitation that this imposes on the size of the data set that can be considered, which would otherwise be restricted to the order of 10,000 points. (In my experiments, 5000 points appeared to be about the limit that my test platform was capable of processing.)

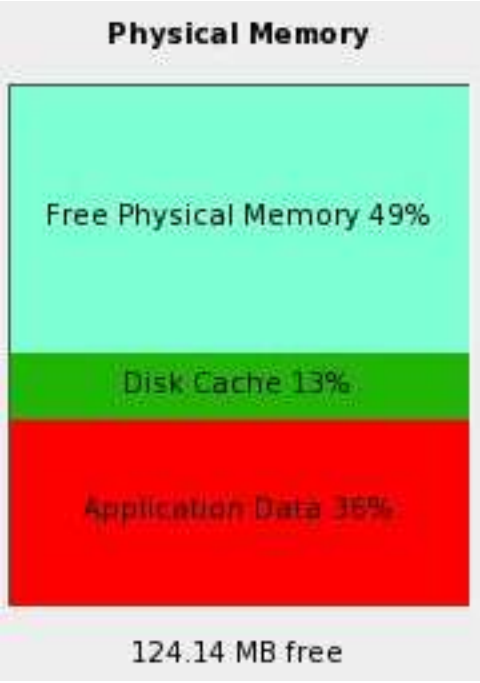

Figure 4: System memory without MDSteer++ running

Figures 4, 5, and 6 show the breakdown of memory usage on the test platform in each of 3 configurations: without MDSteer++ running, with MDSteer++ running on a 2000-point data set, and with MDSteer++ running on a 5000-point data set. Clearly, to run on data sets larger than this, it is necessary to offload the distance matrix from main memory to someplace else.

3.1 Distance Matrix in a Regular File

Storing the distance matrix in a regular file turned out to be reasonably simple, as long as care was taken to do it efficiently. The low-end Linux machines that were used as test platforms had reasonable-sized local disks with 3 GB of free space for the partition on which `/tmp` was located. Because these distance matrices are not needed beyond the end of an MDS run, `/tmp` was an ideal place to locate them. Being on the local disk would also mean it would be much faster than any network-mounted disk.

Linux has convenient functions to create filenames in existing directories in which to place temporary data without worrying about clobbering an existing file with the same name. The other important feature which had to be coded was the ability for random access. The `lseek()` function in Linux provided the necessary ability to jump to any location in the file (as if it were a simple array in memory) and read or write the appropriate number in that exact location.

The size of the file was the size (as determined in Equation 2) multiplied by the size of a data element. MDSteer++ uses the C/C++ type `double`, a double-precision floating point type variable which is 8 bytes long, for its distance values. The 2000-point data set required a 16 MB distance matrix file, and the 5000-point data set required a 100 MB distance matrix file.

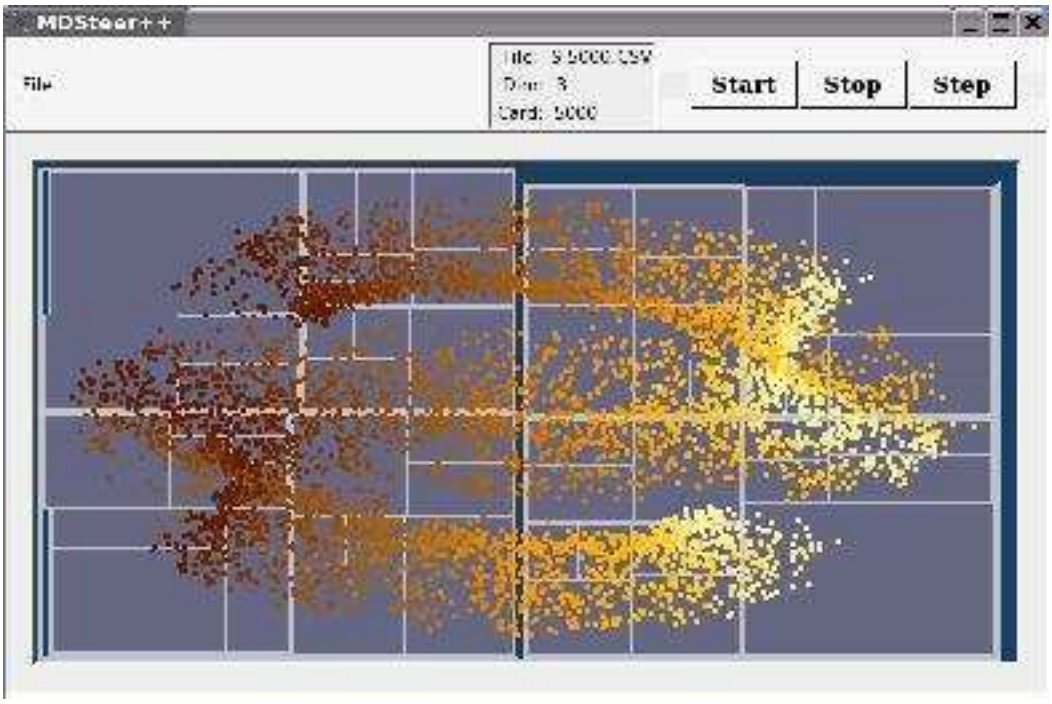

Figure 3: Screen shot of MDSteer++ on a 5000-point extruded S-curve data set

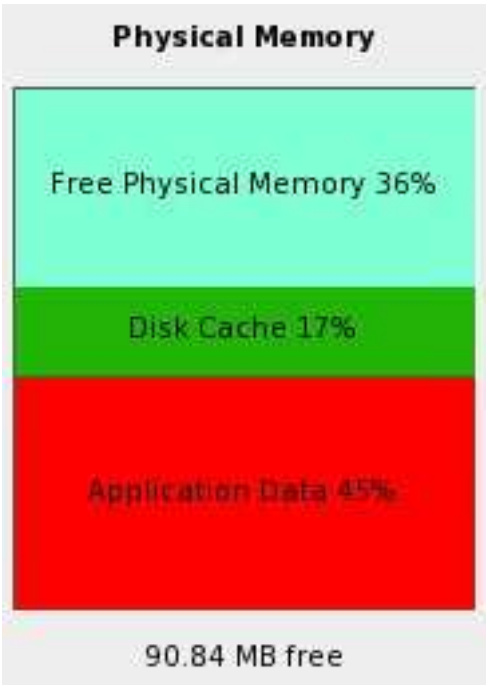

Figure 5: System memory with MDSteer++ on a 2000-point data set

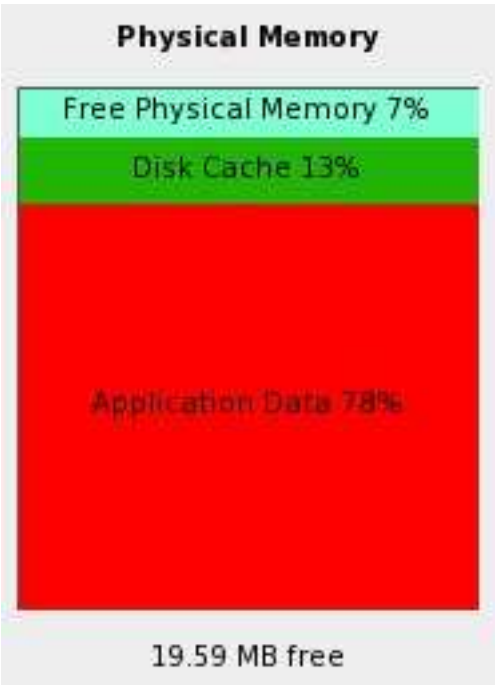

Figure 6: System memory with MDSteer++ on a 5000-point data set

3.2 Distance Matrix in a Database

Even though storing the distance matrix in a regular file provides a measure of offloading, the regular file is being stored on a local disk on that same computer, which could be caching it as well. So in effect, it's possible that the above test might have no real difference with storing the distance matrix in regular memory. But if the data were definitely offloaded from the test platform machine, then there would be a real difference.

As in the previous section, the database would contain temporary information, useful only during the MDSteer++ run. Thus the constructor was designed to CREATE the appropriate database table, and the destructor was designed to DROP it. Random access was not difficult because that is exactly what the SQL database operations SELECT and UPDATE are for. (SQL commands are customarily capitalized.)

Storing the distance matrix in a database was an enterprise fraught with a series of roadblocks and plans to circumvent them.

1. Once the database server daemon had been established on an appropriate machine (other than the test platform machine), attempts to connect to the database over a network were met with failure. Attempts to resolve that were unsuccessful in the available time.

2. If you can't beat 'em, join 'em. I tried running MDSteer++ on the database machine itself, but that was unsuccessful because the database machine doesn't have the appropriate X Windows libraries to allow it to run an X process for display on a remote machine. Moreover, it would not have been a particularly good test anyway because again the database functionality would be running on the same machine as the MDS algorithm.

3. Don't display the MDSteer++ X window. I tried disabling all the windowing functionality in MDSteer++ just to see if it would do the appropriate back end calculations in a GUI-less environment, but the system's front and back ends are too integrated with each other to accomplish that in the available time frame. For the morbidly curious, the attempt at disabling windowing functionality included simulating all the back-end MDS calls, which would normally be made from a method of the GLBox class (a GUI class which could not be instantiated), by calling them from a new static method in that same class which would get around the need for an object of that class to be instantiated.

In pursuing this avenue further, the best course would be to continue attempts to access the database over the network as indicated in item 1. Running the database on the same machine as MDS would interfere with tests of MDS scalability on that machine, and if only the back end is run and appears to work, there is no way to observe the results to confirm successful operation.

4 Results

The testing platform I used was a low-end Pentium III running at 1 GHz, with 256 MB of RAM, running Linux. On this platform, I ran MDSteer++ on the 3-D extruded S curve data set, with cardinalities of 2000 and 5000. When testing scalability limits, using an austere computer is advantageous because then one can more readily observe performance degradation on smaller data sets.

4.1 Distance Matrix in a Regular File

On a 2000-point set, the performance was comparable to that of the system with the matrix in memory. I am uncertain as to whether the file system is just that efficient, or whether the file was being cached in memory, which would effectively render this approach similar to simply storing the matrix in memory and swapping as necessary.

On a 5000-point set, the performance was comparable to that of the system with the matrix in memory, until the algorithm got to the point where about 2000 points had been placed and 3000 remained to be placed. At that point, performance dropped off precipitously, and I terminated the runs before completion. For example, the first 2000 points would be placed on the order of minutes, but the next 200 would require about 2 hours to be placed. What exactly was happening that would cause the system performace to degrade so significantly at that point remains to be determined.

4.2 Distance Matrix in a Database

If the tests using the distance matrix had been successful, it is highly likely that the database overhead, network communication, slowness in the database server, and slowness in NFS (if used) on the database server would cause significantly slower performance of MDSteer++. Nonetheless, this is an interesting avenue to pursue, even if only to verify that.

5 Code Changes

I made the following modifications to the MDSteer++ code base in my local copy of the code. Some of these changes should be checked in to the version on Sourceforge, while others are of a more *ad hoc* nature and should undergo more refinement before being checked in. The transition of the code base from being a research product to being a system for dissemination and use by others is a significant one. Conveniently, I can look to my own experience in learning this code for insight into how best to document and structure it to maximize understandability for other users like myself, where I imagine myself to be somewhat representative of the target audience.

- I made some necessary changes for MDSteer++ to build and run on Linux. These changes should be appropriately integrated so that a single code base can be used on either Windows or unix platforms.

- I fixed the DistanceMatrix constructor to properly initialize the `size` variable.

- I added logging output to the following functions to better observe the volume of data that MDSteer++ is using:

  - In the DistanceCalculator constructor, I added logging output to show the number of items in the ArrayDistanceMatrix and the number of items in the matrix as reported by its getSize() function (to confirm that my bugfix worked).

  - In the ArrayDistanceMatrix constructor, I added logging output to show the number of items in the array and the size of the array in bytes.

- I added a pair of files, FileDistanceMatrix.[h|cpp], following the format of the existing ArrayDistanceMatrix files. These new files perform the DistanceMatrix operations but store the data in a file instead of in an array in main memory.

- I added a pair of files, DatabaseDistanceMatrix.[h|cpp], following the format of the existing ArrayDistanceMatrix files. These new files perform the DistanceMatrix operations but store the data in a database instead of in an array in main memory.

- I changed the structure of the USE_DISTANCE_MATRIX and USE_HASH_TABLE_MATRIX constants. The former is now a numeric value from 0 to 3, rather than a boolean, and the latter has a numeric value of 2. I also added 3 new constants, USE_NO_DISTANCE_MATRIX, USE_ARRAY_DISTANCE_MATRIX and USE_FILE_DISTANCE_MATRIX with values of 0, 1, and 3 respectively. I also changed all references in the code to these constants (specifically in DMDataItemCollection.cpp and DistanceCalculator.cpp), so that USE_DISTANCE_MATRIX is now a constant that is equal to one of the other constants, depending on the mode of operation desired. Note that SpringModel.cpp did not need to be changed because the numeric use of USE_DISTANCE_MATRIX (0=no, [1..3]=yes) does not diminish its suitability for use as a boolean value.

6 Further Research

The most immediate avenue of further research would be to continue trying to examine how the software behaves with the distance matrix stored in a database. Even though overall performance would likely be much slower because of the database overhead, it should allow even low-end computers to run MDS on large datasets without exhausting any resources except time and the patience of the user.

Since MDS preserves high-dimensional large-scale features such as clusters in the low-dimensional space to which it maps the points, one might ask whether and to what degree smaller-scale features are preserved. For instance, if the cardinality of the data set is very high, high enough that the set can be observed to have fractal characteristics, would the MDS technique transfer some of those characteristics to the lower-dimensional space as well? What other geometric features could be mapped to the low-dimensional space through MDS?

What happens in MDSteer++ when the user doesn't steer anything? It is similar to the behaviour observed by Chalmers *et al* in their work? Are the time complexity and residual stress similar to those observed by either Chalmers *et al* or Jourdan and Melançon. To the degree to which the MDSteer++ algorithm is different than those in the previous works, what can we say about those differences and similarities?

7 Further Code Development

There are a number of things that could be done to make MDSteer++ more appropriate for widespread release.

- The system currently specifies the entire UI in the .cpp files. Those commands should be abstracted out into .ui files such as would be created with QT Designer. This would make the overall GUI layout much more clear and easier to modify, and would draw a nice line between basic GUI elements and the code that is written to deal with those elements for the purposes of this application.

- A much clearer line should be drawn between the MDS functionality and the UI components. For instance, a function that deals with the percentage of anchor points should not be part of the GLBox class; the former is back-end functionality and the latter is UI functionality. This increased modularity could result in MDSteer++ providing not only an executable but also one or more libraries which others could link in to their software to help satisfy their own MDS needs.

- In general, the system should be made as modular as possible. North and Shneiderman have suggested a model of infovis tool development in which users can construct more complex visualization tools by 'snapping together' tool elements in their own customized ways.[8] This is an excellent idea, but it should be taken one step further and applied to components of visualization tools in general, whether back end or front end. An MDS engine should not be married to a particular interface, and by separating these components and potentially creating a set of protocols for communication between the front and back ends, it may be possible to realize the goals of "Snap-Together Visualization" in broader ways than North and Shneiderman have suggested.

- MDSteer++ should have a mode allowing it to run on a data set without user input or GUI components. This might result in a reversion to the previous MDS algorithms of Chalmers *et al*, or could be something new and different.

- The DISTANCE_MATRIX constants mentioned previously should be replaced so that it is not necessary to recompile the source code in order to try using different kinds of distance matrix storage. Ideally, it should be possible to create a single executable for a particular platform and be able to distribute that, and make recompilation only necessary for those who wish to play with the source code themselves.

- The `moc_*` files are automatically generated and should not form part of the set of checked-in files for this project.

- The code should be cleaned up so that it does not generate compile-time warnings.

- The changes necessary to make the code compile on Linux should be carefully integrated so as not to damage the ability of the code to compile on Windows. In addition, it would be worthwhile to see what if any changes would be required for the code to compile on a Mac OSX platform, in order to achieve broad cross-platform operability with a single code base.

- I am highly curious as to whether and by how much an MDS implementation in Python would be slower than the current one in C++. If the efficiency hit is not too severe, it might be very interesting to reimplement this system in Python, especially if some restructuring of the code base is necessary anyway in order to achieve some of the desired modularity and other goals stated above. The resultant code would likely be more portable than the current C++ implementation is, and have a much smaller code base in terms of lines of text. But the biggest advantage is that if the code base is appropriately structured, some modularity comes for free because Python scripts can be simply and natively run at any level without the need to construct specific libraries or executables.

8 Conclusion

In this project, I have examined some scalability issues in multidimensional scaling in general and in the MDSteer++ software system in particular. Previous research on MDS by a number of authors has provided a good context in which to consider scalability and efficiency in MDSteer++. I have been successfully able to offload the distance matrix that MDSteer++ creates in order to more efficiently perform its MDS calculations. However, questions remain as to how effective a test this is because operating system considerations such as caching and swapping could interfere with these tests. Offloading the distance matrix to a database would get around these issues, but likely at a signficant cost due to network communication, database software overhead, and other factors. Nonetheless, this avenue of study will be continued, and tests remain to be done

to see how MDSteer++ performs with the distance matrix stored in a database. Finally, I am excited about the prospect of continuing the work needed to get MDSteer++ ready for prime time use by others interested in MDS. In the CPSC 533C class project presentations, I was surprised by how many other projects used spring models and data sets that could make use of MDS. If MDSteer++ is made sufficiently useful and easy to use, it could be used not only by those interested in MDS *per se* but also by anybody who could benefit from using MDS, purely as a user, in their own work.

REFERENCES

[1] G. Ross A. Morrison and M. Chalmers. A hybrid layout algorithm for subquadratic multidimensional scaling. In *Proc. IEEE Symposium on Information Visualization*, pages 152–158, 2002.

[2] G. Ross A. Morrison and M. Chalmers. Fast multidimensional scaling through sampling, springs and interpolation. In *Information Visualization*, pages 68–77, 2003.

[3] Authors Unspecified. *A Guide to Using MDSteer++ Alpha Release 0.5*, February 2005.

[4] M. Chalmers. A linear iteration time layout algorithm for visualizing high dimensional data, 1996.

[5] T. Munzner D. Westrom and M. Tory. Progressive binning for steerable multidimensional scaling. 2005.

[6] F. Jourdan and G. Melançon. Multiscale hybrid mds. In *Intl. Conf. on Information Visualization (London)*, pages 338–393, 2004.

[7] A. Morrison and M. Chalmers. Improving hybrid mds withi pivot-based searching. In *Proc. IEEE Symposium on Information Visualization*, pages 85–90, 2003.

[8] C. North and B. Shneiderman. Snap-together visualization: Can users construct and operate coordinated views? November.

[9] Allan Rempel. Fast progressive transmission of images using wavelets with sorted coefficients. Master's thesis, University of British Columbia, 1997.

[10] G. Ross and M. Chalmers. A visual workspace for hybrid multidimensional scaling algorithms. In *Proc. IEEE Symposium on Information Visualization*, pages 91–96, 2003.

[11] M. Williams and T. Munzner. Steerable, progressive multidimensional scaling. 2004.