Algorithms and data structures for graph analysis

Introduction

Software implementations for graph structures can represent a graph as an adjacency list or a sparse adjacency matrix. There are no standards regarding the implementation details of adjacency list or adjacency matrix structures. Notable parties propose different implementations for adjacency lists [1]. Guido van Rossum suggests using a hash table with arrays, where the arrays indicate edges to vertices. Cormen et al. [2] suggest using an array of singly linked lists. Goodrich and Tamassia propose dedicated objects for both vertices and edges, allowing for easily adding information to both vertices and edges. Each of the implementation suggestions has advantages and disadvantages. One can easily come up with more variations, but their trade-offs in the context of graph applications are not clear.

Typically the choice of implementation will be dominated by memory/ time trade-offs and the need for dynamic graphs, i.e. the need to alter a graph at random. The size of available datasets is ever increasing. Although big-data environments allow for massive parallelization, not every graph application can be parallelized; this puts these trade-offs more on the foreground. Behind these observations there is a central question that is relevant to the practice of graph theory: what is the price of using a general framework for creating graph applications? Is it best to think of designing a graph application in terms of a choice between one of the general frameworks (NetworkX , SNAP, Jung, etc.) Or should one see the design of a graph application more as a detailed choice of a set of applicable data structures? A quick answer is: ‘trust the experts’. The expert answer will of course always be: ‘it depends’. This question is relevant for dedicated applications where more performance means more competitive advantage.

In this project the impact of data structures on the performance of graph applications is assessed. Both the impact on artificial graph generation and the impact on several graph algorithms will be assessed. These two steps provide a minimal workflow, applicable to many situations. More specific the impact of data structures will be assessed for the following tasks:

- Synthetic graph generation using a nearest neighbor algorithm [ref paper]
- Spectral clustering of a graph [3]
- Calculating maximum flows using the Ford-Fulkerson

Document overview

This document contains the following sections:

- A critique motivating the research subject
- A short overview of relevant data structures with some characteristics
- A rudimentary performance measurements of relevant data structures in the context of nearest neighbors
- A memory optimized data structures for graphs
- A comparison of three data structures in a graph context
- Performance measurements of synthetic graph generation using a nearest neighbor approach
- Performance measurements of a spectral clustering algorithm using different data structures
- Performance measurements of the Ford-Fulkerson algorithm using different data structures

Critique

A review of major graph implementations shows that implementation decisions regarding data structures are not made explicit. NetworkX [ref website], an elaborate Python based framework, does not mention implementation decisions regarding graph data structures; nor do SNAP [4] or Jung [5]. The primary reason for this is likely to be the fact that all these frameworks are positioned as general frameworks. The qualification of general in this context can be interpreted as ‘flexible’ or ‘broad, without specific aim’. The frameworks aim to offer the broadest support for graph applications, including dynamic graphs. The inclusion of dynamic graphs most likely precludes (or not?) the application of for example static arrays. A more flexible data structure for storing nodes like a maps or ordered maps seems more likely. But then the representation of edges remains open: linked lists, or dynamic lists, like Python lists or C++ vectors, or also a map or a set?

In computer science the performance characteristics of different data structures are well known and stressed. Performance characteristics of specific algorithms like for example breadth first search or depth first search are also generally available. The performance trade-offs of data structures in the context of graph applications seem less considered, and therefore less known. The choice of data structure is of course impacted by graph algorithm under consideration; but then most graph applications are written in the context of general frameworks: this completes a circle.

As noted before, the trade-offs between data structures become more relevant as the size of available graph datasets grows, and the interests in a competitive advantage are growing. Even if a graph algorithm itself is not parallelizable, one could for example tap into the power of map reduce frameworks to pre-sort datasets, off-loading parts of non-critical work; this is often done in information retrieval work flows. This project aims to

assess the trade-offs between data structures for graph applications, with a bias towards applications for larger data sets.

First an overview of relevant properties for graph data structures is given.

Relevant properties for graph data structures

The following general properties can be listed as relevant for graph data structures:

- Fast random access to nodes and edges
- Fast iteration over nodes and edges
- The ability to add and remove nodes and edges
- Memory efficiency

Typically one would like to lookup items in constant $O(1)$ or $O(\log n)$ time. Iteration allows for visiting nodes or edges in a single sequence without random access, for example in an iteration context. The ability to remove nodes and edges could be optional in some situations, but should generally be supported. Regarding memory efficiency three notes can be made:

- Representing a node ('0') as the first element in a list and deleting that node would result in node '1' being represented as '0'; this is not favorable. One would require 'sparseness'.
- Storing edges outside the context of
- To efficiently remove nodes for an undirected graph both outgoing and incoming edges should be accessible fast (i.e. fast row and column access)

With the properties above in mind three interesting scenarios emerge:

- Which design is best if optimizing for memory?
- Which design is best if optimizing for adding and removing nodes and edges?
- What are the speed trade-offs?

These topics will be investigated in the next sections. First an overview of candidates for node and edges data structures will be given.

A short overview of relevant data structures

In this section the following six relevant data structures are reviewed:

- (Sparse) matrix
- Linked list
- Indexable skip list
- Dynamic array (vector)
- Ordered map (tree backed)
- Unordered map (hash table backed)
- Judy array

This overview is not intended to be exhaustive, but should cover enough ground. At the end of this section an overview of some key properties of these data structures is given. The data structures will be shortly reviewed in the next sections.

Full matrices provide a lot of good properties as a data structure for graphs: fast insertion, lookup and deletion. Traversing the (non-zero) elements of a row or column though is not so fast since all elements have to be visited. In practice the high memory requirements often prevents application of a full matrix. Sparse matrices circumvent the drawbacks of matrices, at a cost of course. Sparse matrices, like adjacency lists, do not have a fixed implementation. The requirements for sparse matrices mimic the requirements for graph data structures, as mentioned in the previous section (notably fast row and column access). Any graph data structure will in a way mimic a sparse matrix data structure; because of this the properties of sparse matrices will not be assessed to avoid a recursion of the questions at hand. Implementations of sparse matrices can be based on dictionaries of dictionaries, lists of list, coordinate lists (lists with (x, y) or (x, y, value) tuples). The last representation is popular in map/reduce contexts.

A linked list is a chain of value and pointer tuples. Linked lists readily support fast iteration and insertion. Storing the values and the pointers requires extra memory. Since random access is not supported, lookups and deletions require traversal. For this reason ordering the items in the list for example does not help lookup.

Skip lists try to fix these drawbacks. Skip lists add additional pointers to values farther down the chain. Lookups can skip large segments of items by ‘jumping’ over them. Adding multiple forward options to elements down the chain can mimic a binary search structure. By adding the number of skipped items per jump, additionally random access of elements by index is possible. This structure is only introduced because of its fit for one the experiments (graph generation).

A dynamic array is range of continuous memory, containing elements of a specific type (integer, double, etc.). Contrary to a linked list, a dynamic array allow for index lookup, and allow for a binary search in ordered items. Key point is that deleting items requires moving large blocks of memory.

An ordered set or map places elements in a binary tree structure. Lookup, insertion and deletion are reasonable. The tree structure requires a lot of memory Elements can be iterated over in order. A hash map stores elements in an array structure based on the hash value of the element. Collision can be resolved in multiple ways. The elements can not be accessed in order.

Judy arrays apply tries to save memory. The design is very complex. I Imagine an n-ary tree where the intermediate (integer) values are woven into a trie.

Data structure	Insert	Delete	Lookup	Space	sizeof
Matrix	*O(1)*	*O(1)*	*O(1)*	*O(n)*	varies
Linked list (unordered)	*O(1)*	*O(n)*	*O(n)*	*O(n)*	24 bytes
Linked list (ordered)	*O(n)*	*O(n)*	*O(n)*	*O(n)*	24 bytes
Indexable skip list	*O(log n)*	*O(log n)*	*O(log n)*	*Exp. O(logn)*	32 bytes
Dyn. array (unordered)	*O(n)*	*O(n)*	*O(n)*	*O(n)*	24 bytes
Dyn. array (ordered)	*O(n)*	*O(n)*	*O(log n)*	*O(n)*	24 bytes
Ordered map (tree)	*O(log n)*	*O(log n)*	*O(log n)*	*O(n)*	24 bytes
Unordered map (hash)	*Exp. O(1)*	*Exp. O(1)*	*Exp. O(1)*	*O(n)*	40 bytes
Judy arrays					

In the next sections some rudimentary performance measurements of relevant data structures is presented.

Rudimentary performance measurements of relevant data structures

Rudimentary speed and memory characteristics of the previously mentioned data structures are assessed in this section. The following characteristics are assessed:

- Memory consumption (figure 1)
- Creation speed of data structure (figure 2)
- Iteration speed over data structure (figure 3)
- Lookup speed of random elements in data structure (figure 4)
- Deletion speed of elements in data structures (figure 5)

During measurements sets were used rather than maps to accentuate the overhead of index structures rather than extra dictionary values. This was not possible for Judy arrays.

During the experiments the C++ wrapper of Judy arrays required lots of debugging. Creating a larger amount of Judy arrays required a lot of memory. For this reason Judy arrays were dropped.

As an alternative to a dynamic array a deque was added to the measurements. A deque supports fast addition of elements to the front and back; this is made possible by not storing the elements in contiguous memory, but in contiguous blocks. This property does not apply in the context of this study, further considerations is drop.

Sparse and dense maps and sets as implemented by Google were added to the measurements in a later stage. These data structures use bins, like

hash maps and groups. Time was not available to investigate the structures in more detail (this is swapped for other activities). The ability to reserve memory upfront indicates that that some type of array structure is involved (probably involving groups/bin of length magic number 48).

Lookups in dynamic arrays (and deques) are based on a binary search in ordered values, not on index access due to reasons of sparseness. Two compare the creation of ordered dynamic arrays using insertion sort and post sort, an additional measurement was performed (figure 6).

Due to brevity reasons the results will not be discussed. Based on the results two candidate data structures will be presented in the next sections.

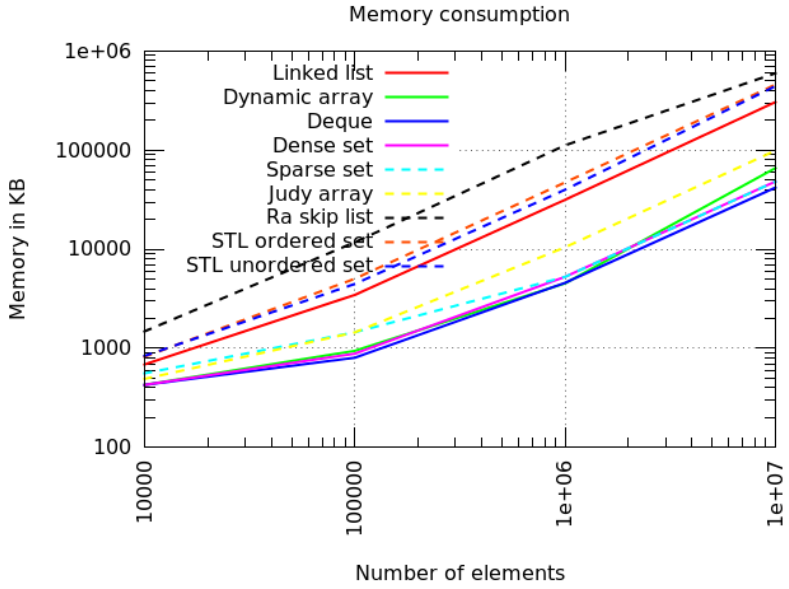


Figure 1 Memory consumption

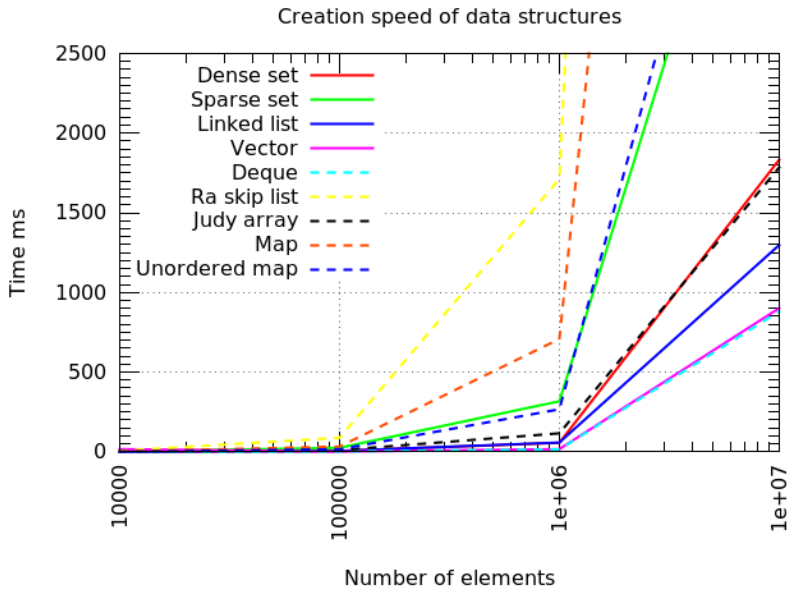


Figure 2 Creation speed of data structures

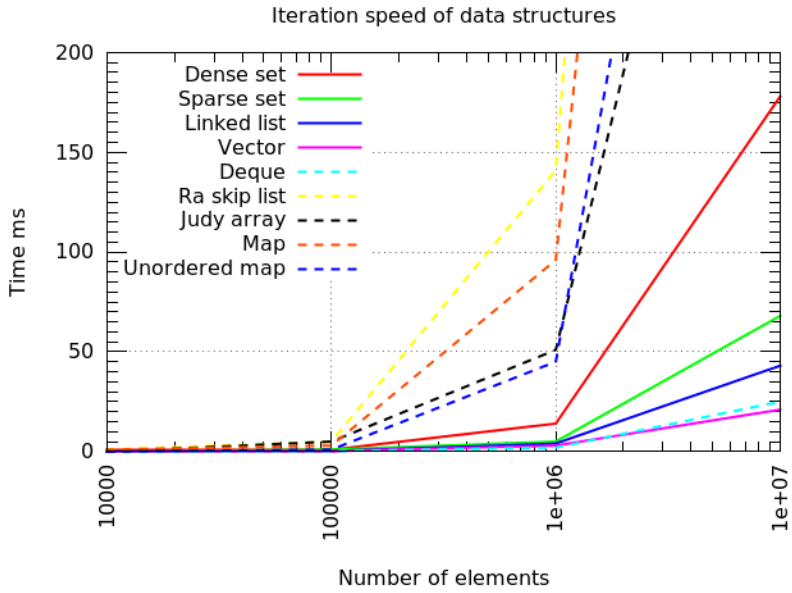


Figure 3 Iteration speed of data structures

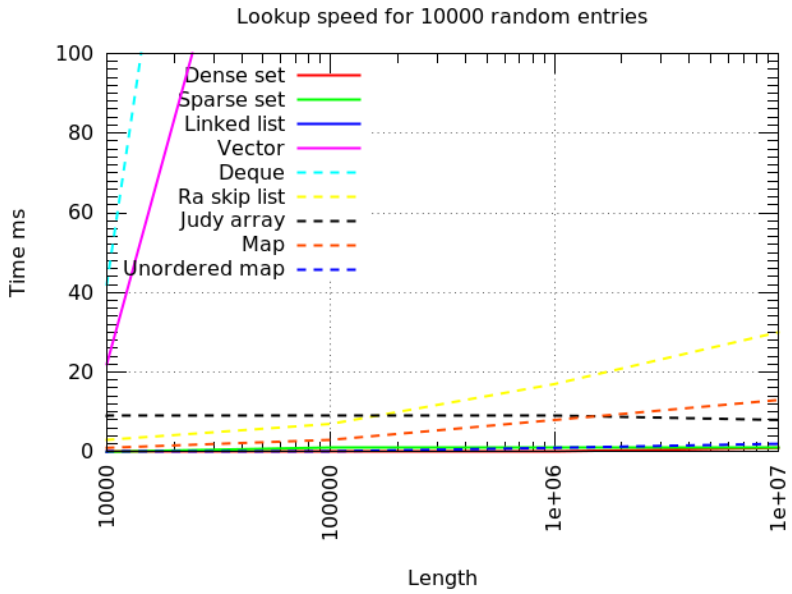


Figure 4 Lookup speed of data structures

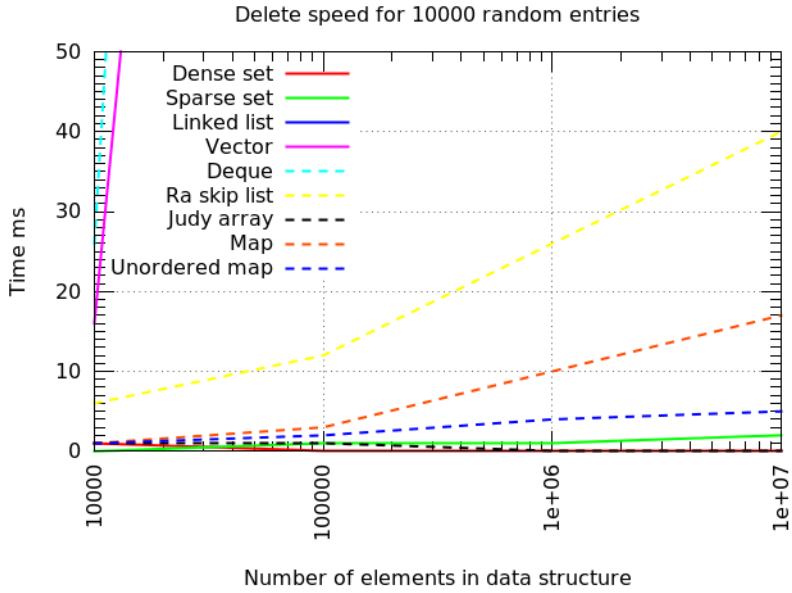


Figure 5 Deletion speed of 10000 random entries

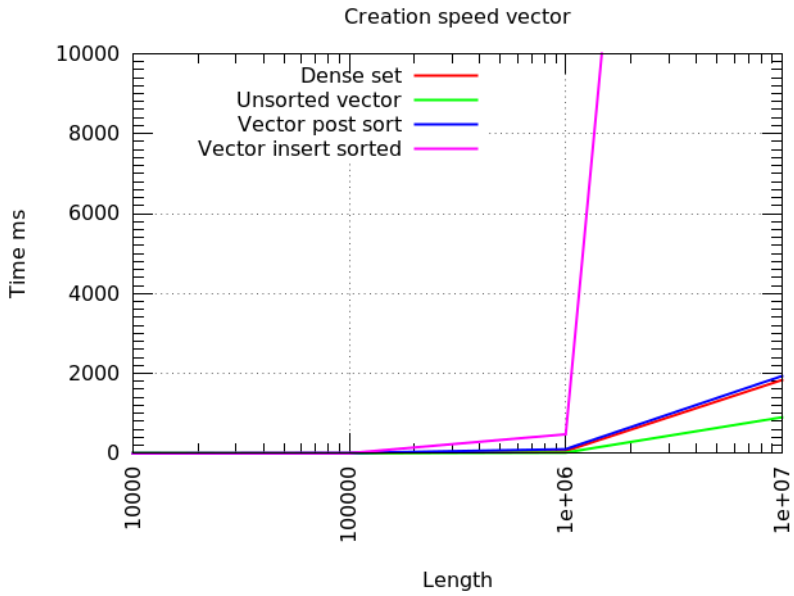


Figure 6 Creation of ordered dynamic arrays (dense set added for reference).

Tooling

For this study the following tools were used:

- Ubuntu 14.04
- C++/gcc
- Workstation Xeon 12 core, 2.7 Ghz, 24Gb RAM, 6Gbps disk

A memory optimized candidate for data structures

This section is dense as it involves the synthesis of a lot of information. For reasons of simplicity directed graphs are considered. First the memory optimized candidate is presented, second a short note on a read/write optimized candidate.

Memory optimized candidate

As the previous measurements show, dynamic arrays are efficient in storing data; as efficient as it gets. Some initial observations are:

- In the memory measurements there seems to be some memory overhead, this is likely some fragmentation that is not immediately cleaned up; it will be ignored.
- Node id-s can be assumed continuous (ignoring deletions) as it can be deferred to off-line processing.
- Insertions into and deletions from vectors of moderate length (10.000 – 100.000) is still moderately possible (this range is what one might call 'Pythonistan')

- Using primitive types to store data in dynamic arrays prevents C++ object structure overhead
- If power laws are involved then using a separate vector to store the edges of a node might present unwanted overhead.
- The 'sparseness' argument requires at least storage of the node id-s itself apart from storing the edges.
- If storing very large graphs the allocation of very large contiguous memory might cause memory management issues.
- For the candidate deletion of nodes and edges is not considered optional, since this would limit the application space.
- Applying variable byte encoding like in search application is not favorable since indexed access is required
- The sparse hash map can store elements with an overhead of one bit
- Lookups in sorted vectors using binary search is relatively slow

Based on the observations above, the following progression of memory optimized data structures can be made.

A simple data structure would store the node id-s in an array, and store the edges of the nodes in separate arrays (one per node, the edges are sorted), accessible via a central array of pointers to those arrays. If creating a vector per node for the edges is to much overhead then the following progression can be made.

The edges are not stored in separate arrays, but are stacked by order of node id in a single array. Since this could result in a huge array for the edges, a further progression can be made to help memory management.

Based on the node id-s bins are made using the modulo operator. The node id-s are stored in sorted arrays, one per bin. These arrays are accessible via an array of pointers, which length is based on the modulo argument. The edges are stored sorted in a separate array per bin and stacked in order of the node id-s. A third array (per bin) stores the starting index of the nodes edges (i.e. the third node in the bin is '5', the edges of which start at index '45' of the edge array). Since nodes have to be looked up in the node arrays using binary search, further refinement is needed, as the measurements indicate.

Instead of using a separate array for the node id's and the edge indexes, a single sparse hash map (SHM) can be used to store both the node id-s and the edge index at once. As the measurements show, this is fast and efficient. Additionally the edge array can be discarded. Since internally the nodes in the sparse hash map are not stored continuously, binning is not necessary for nodes. The edge array though can still be broken down in bins: this is likely to be the largest structure.

Insertion of nodes would require appending to the sparse map; this should not be an issue.

Insertion of edges would require updating the array with the edge indices, since the the edges are stacked per node. This is not efficient. But in one case though:

- assuming more or less continuous node id-s
- assuming a moderate range of edge degrees
- setting the modulo argument to the number of nodes

it is possible to land in 'Pythonistan' since every node has it own edge list. On appending a node:

- the modulo argument has to be incremented
- an extra edge edge array has to be created

This is not an ideal situation, but might be manageable.

Deletion requires a deletion from the node in the sparse map (fast as the measurements show) and a deletion from the edge arrays. Deletions from the edge array could be supported reasonably if the edge array is stored in in sparse hash map too and the reverse edges are stored for fast incoming edge lookups. This would probably need a 'landing edge' that is never deleted for edges in the edge array. Random edges lookup can not use binary search to lookup random edges if the edges are stored in a sparse map; this possible if each node has a separate sparse hash set: this requires more memory but might be an option for the speed optimized candidate.

The resulting data structure allows optimization under several circumstances and supports a wide range of applications. In coming section the properties of this structure will be assessed. Interestingly the structure is some form of a map of maps.

Speed optimized candidate

Additionally the notion of speed is less uniform than the notion of memory. In light of the results in the previous section, the rest of this project the memory optimized candidate will be compared to SNAP instead, making adjustments if necessary.

A comparison of data structures in a graph context

In this section the memory optimized candidate (MOC) is benchmarked to SNAP for several graph related tasks. First a table of results is presented, followed by some comments. The measurements are intended to test the data structure requirements as stated earlier, and might reflect general best practices.

Structure	Task	Dataset	Results	Comp.
SNAP	Read data set	LiveJournal	48s	
MOC			6.3s	0.13
SNAP	Memory	LiveJournal	1854MB	
MOC	1 bin		300MB	0.16
MOC	2^14 bin		302MB	
	reverse		+300 MB	
SNAP	Lookup 1M random nodes	LiveJournal	0.11	
MOC	Node arrays+bin. search		84s	
MOC	Nodes in SHM		0.12	1.1
SNAP	Power iteration	LiveJournal	20.5s	
SNAP	Native using Lanczos		10.08	
MOC	1 bin		13.6s	0.66
SNAP	Remove 1M nodes		-	-
MOC	14^2 bins		-	-
SNAP	Create BFS tree	LiveJournal	**	
MOC			6.7s	0.56

The datasets used were taken from SNAP [5]:

- LiveJournal social network (4847571 nodes/68993773 edges)

Since the MOC does not build up any binary tree structures (BTS) in memory for storing the nodes, reading datasets is very fast. Two preprocessing tasks are required though for this speed:

- a list of unique node id-s with edge counts (a preprocessing task)
- the input file consists of edges sorted per node (a preprocessing task)

Note that the graph is directed and that no reverse edges are stored.

In the MOC memory measurement the measured quantity was DRS memory.

The power iteration shows that the MOC is fast enough iteration wise. Note that the SNAP power iteration was a simple one, created for this measurement. The SNAP power iteration iterated over the edges, the MOC power iteration iterated over the nodes first. This is a necessary step that does not seem to hurt. Native Lanczos in SNAP is faster.

Removing edges is not implemented due to time contstraints.

The creation of the BFS tree can be done fast in MOC by avoiding the usage of set like structures. First an appropriate empty node and edge structure is copied were the node id-s are marked using a special marker (MAX_INT for example). With the sparse representation this is not too bad, it requires about as much memory as a set containing a 95% SCC (as can be concluded from the measurements, and there might be some logic behind this), but it is considerably faster. Next during the BFS each marker is replaced by the number of valid outgoing edges, if the node is visited. If a node has been visited, the edge is marked (MAX_INT for example), indicating is has been surpassed. Finally the node structure is compacted.

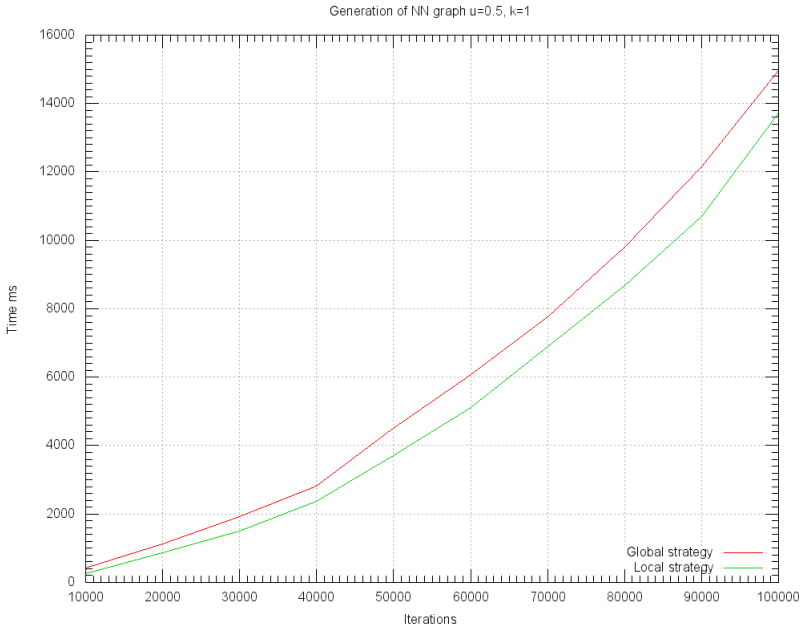
Intermediary notice

Up to this point the project has been updated after the milestone. The next results have not changed since the milestone, and are presented as is; the conclusion exempt.

Performance measurements of synthetic graph generation

In this section the performance characteristics of synthetic graph generation are assessed. To generate the data a nearest neighbor approach as described by [ref] is used. A critical part of the generation process involves randomly choosing a potential connection from a large set of potential connections. Two strategies are explored: storing all potential edges in one set (‘global’), and storing potential edges as ‘local’ sets connected to the ‘central’ node of the potential connections. In light of the previous measurements, nodes are stored as elements of vectors since during graph creation nodes are ‘append’ only. For storing the potential connections random access skip lists are used, since this structure seems to fit the requirements best. This might come with a hefty memory penalty. Note that although maps allow for faster lookups and deletions, these structures do not allow for indexed lookup, which is favorable for random draws.

In the simulations u=0.5 and k=1.



Outside the moderate range, memory/speed trade offs are unavoidable, the choice of data structures have a big impact on the performance of the graph generation algorithm. The trade-off to be made would likely be between random access skip lists for speed and vectors for memory efficiency. But this only relevant if the number of potential edges exceeds the moderate range; for a lot of cases using for example vectors to store potential connections can suffice. Using more specific knowledge of u one might be able to optimize memory requirements by using a local strategy and storing either the unconverted potential connections or the complement.

Performance measurements of spectral clustering

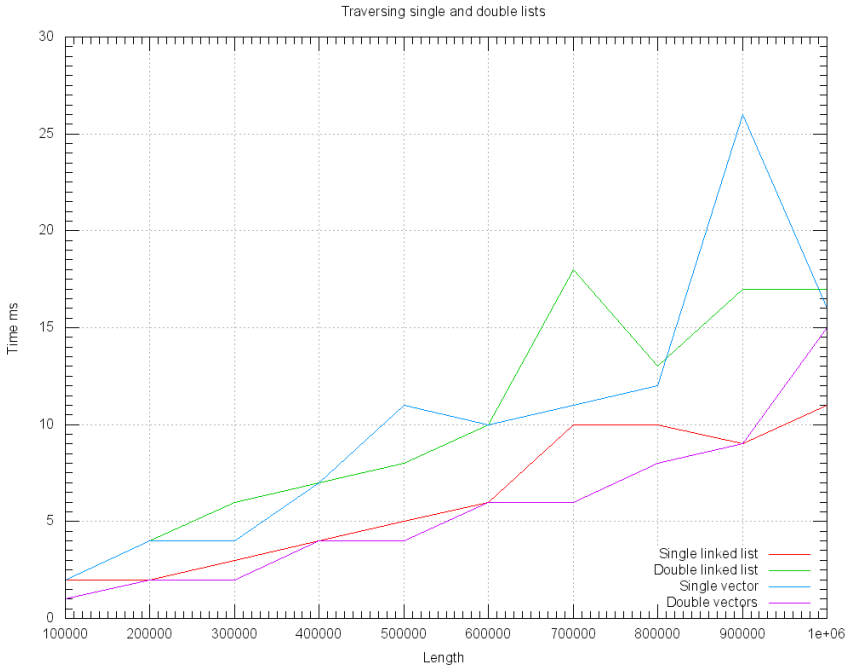
One can apply spectral clustering using the power method to find the eigenvectors of a Laplacian matrix representation of a graph. Let it first be noted that map reduce is specifically fit to perform power iterations over large graphs. This fact greatly diminishes the applicability of performance assessments in this section.

In this assessment the NJW [3] method will applied, using the largest eigenvector. After the eigenvectors are found, the nodes are clustered using a k-means algorithm initialized with kmeans++.

The nodes are stored as vectors. For the iteration of edge values one must then store both a column index and the value itself. Four implementation options seem valid for storing the edges:

- A linked list of column-value pairs
- Two parallel linked lists with column and values
- A dynamic array of column-value pairs
- Two parallel dynamic arrays with column and values

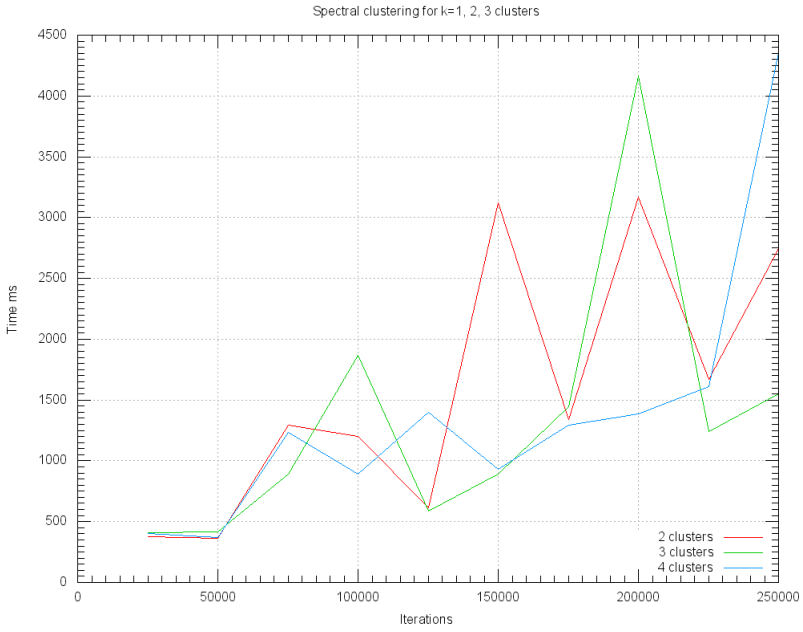
A very small traversing experiment shows that the last structure is the dominant choice. i.e. both speed and memory wise. Since maps also allow for iterator access, the performance of maps is considered equal to the performance of a linked list with objects. This should ne comparable to the single linked list implementation; the performance of which is comparable to the data structure of choice. Hence, a maps-of-maps generic graph structure is likely to yield similar performance (requiring more memory of course as shown earlier).



Crunching two parallel dynamic arrays set-up a bit more, the final data structure for the assessment contains just three lists:

- One list with the cumulative number of edges over the nodes
- One list with block wise the edges per node (addressable using the first list)
- One list with block wise the values per edge in the same order as the second list.

A power iteration now involves the iteration over three consecutive memory blocks. This sequential pattern shows that the computation can be performed parallel, using for example map/ reduce. A similar graph generation set-up is used as in the previous section.



Performance measurements Ford-Fulkerson

In this section the impact of data structures on the performance of the Ford-Fulkerson maximum flows algorithm is assessed. Unlike the algorithm in the previous section, this algorithm is not as easily parallelized; which provides an opportunity for improvements.

Two different implementations of the algorithm will be compared. The first version uses as data structure a map of maps to represent a weighted graph, the second version uses the compact data structure with three vectors as mentioned in the previous graph. The algorithm requires the random access on the edges of a node. As shown at the beginning, one would expect maps to be faster.

Implementation	Iterations	Nodes	ms
Maps of maps	10000	6	760
Three vectors	10000	6	70

A non-representative micro experiment shows that solving a small 6 nodes flow network 10000 times is actually done faster by the graph with three vectors. Clearly the seek time to assess the edges weights is not dominating the time taken yet. The probability that this result is real is very low. Even if proven valid, generalizing the results should be done with utter care.

Conclusion

In the introduction it was stated that textbook implementations leave ground uncovered. In this project it was shown that newer data structures (notably from Google) offer different possibilities, that allow for the design

of new solutions. These designs are relevant in contexts were performance delivers an competitive advantage.

It was shown that dynamic arrays do a decent job in the moderate range; dynamic arrays must have been very well optimized on all levels. The point above explains much of the success of Python and its list structures. It seems that there new opportunities in ‘extremistan’ trade-offs are hard and shall be made consciously.

C++ turned out a good type of environment for this kind of assessment; though implementing software in C++ is not trivial. The information presented in this context is not new; accessing this information requires extensive search, and evaluation.

Project statistics

C++ code: 1400-1600 lines
Python POC code: similar to milestone (600-700?)
Gnuplot code: 200-300 lines

A short note on a bumpy road

These project results are at the end of a bumpy road in many ways. Implementing software in C++ proved hard; harder than in Python for example. C++ might not have been the right choice productivity wise. The language does provide opportunities, as was hopefully shown.

Looking back at the midterm expectations I can conclude the following. Implementing the software so far proved harder than expected. But finding new technical opportunities and molding these into a design is hard, especially of the beaten track. I had a hunch of this, but not strong enough. I underestimated the time involved for the extra requirements as presented by the TA. These requirements fall more into the space of graph analysis itself. I can see the reasons for the requirements, and would have liked meeting those. This should in my opinion not diminish the value of the hard work put in and the results: this is the central question though.

This is my seventh course at Stanford. Compared to other courses, I put in as much time for this course as for the other courses. To make progress with the project (and for personal reasons), I took more than four weeks leave, and even ended up terminating my contract. As stated, a bumpy road. Time ended up being available by setting priorities. From my part I am happy to have taken the challenge. Compared to the other courses, in this course I reached far to get into more serious space regarding bigger data crunching.

References

[1] “Adjacency list” *Wikipedia*. Wikimedia Foundation, accessed 10 december 2014.

[2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*(2nd ed.). McGraw-Hill Higher Education, (2002).

[3] Ng, A., Jordan, M., Weiss, Y.. On spectral clustering: analysis and an algorithm. In: Dietterich, T., Becker, S., Ghahramani, Z. (eds.) *Advances in Neural Information Processing Systems* 14, pp. 849- 856. MIT Press, Cambridge (2002).

[4] Leskovec, J. *Stanford analysis project*. Stanford University, accessed 10 december 2014.

[5] “Java Universal Network/Graph Framework”. Sourceforge.net, accessed 10 december 2014.