A comparison of search times on compressed and uncompressed graphs

TIM GRANSGKOG AMANDA STIGÉR
A comparison of search times on compressed and uncompressed graphs

TIM GRANSKOG
AMANDA STRIGÉR

Degree Project in Computer Science, DD143X
Supervisor: Danupon Nanongkai
Examiner: Örjan Ekeberg

CSC KTH 2015-05-08
Abstract

This report researches whether it is possible to speed up search algorithms on general graphs by means of graph compression. To determine if this is possible the compression methods RPGraph, LZGraph, DSM and $k^2$-tree were used in combination with the breadth first search and depth first search algorithms. The search algorithms were run on graphs of different sizes, both in compressed and uncompressed form, and the run times were compared. The results showed that compression adds too much overhead in neighbour list retrieval times to be able to reduce search times when both the uncompressed and compressed graphs can be held in main memory.
Sammanfattning

Denna rapport undersöker om det är möjligt att göra sökalgoritmer snabbare på allmänna grafer med hjälp av grafkomprimering. För att avgöra om detta är möjligt användes komprimeringsmetoderna RPGraph, LZGraph, DSM och $k^2$-tree i kombination med algoritmerna Bredden Först Sökning och Djupet Först Sökning. Sökalgoritmerna kördes på grafer av olika storlekar, både i komprimerad och okomprimerad form. Resultaten visade att kompression ökar tiden för att hämta ut grannlistor för mycket för att kunna minska söktider då både de okomprimerade och de komprimerade graferna kan hållas i primärrinnet.
Contents

1 Introduction .............................. 1
  1.1 Compression Methods .................... 1
  1.2 Problem Statement ....................... 1

2 Background ................................ 2
  2.1 Dense Subgraph Miner ..................... 2
  2.2 $k^2$-tree ................................ 3
  2.3 Grammar-based Compression ............... 4
    2.3.1 RPGraph .............................. 4
    2.3.2 LZGraph .............................. 5
  2.4 Splitting a Graph ....................... 5

3 Method .................................... 6
  3.1 Hardware ................................ 6
  3.2 Compression Methods .................... 6
  3.3 Search Algorithms ....................... 6
  3.4 Test Data ................................ 7
  3.5 Measuring Memory Usage .................. 7
  3.6 Specific Settings ....................... 7
    3.6.1 Compression Method Settings ....... 8

4 Results ................................... 8
  4.1 Average Search Times .................... 8
  4.2 Average Search Times as Related to Density 10
  4.3 Compression Ratios ....................... 14

5 Discussion ................................ 14
  5.1 Method Discussion ....................... 14
  5.2 Discussion of Results ................... 15
  5.3 Future Research .......................... 16

6 Conclusion ............................... 16


1 Introduction

Graph theory is a significant part of computer science and is a well researched area with many practical applications. Because of this extensive research there exists many well-known and proven correct solutions to a lot of real life problems that can be solved by using graphs. A recent example of a problem that can be represented as a graph is the problem of trying to model the Internet to simplify analysis and extraction of relevant information. With the web expanding the graphs used to represent them are growing limitlessly. A recent crawl of the Internet yielded a web graph with over 42 billion links and close to 1 billion web pages with just the document containing the URLs adding up to 688 GB in size when uncompressed [5]. Graphs of this size are far too large to store in main memory and storing them on drives is not an option as it slows down the data mining. Web graphs is a specific type of graph and there exists other types of graphs used to model problems and these can also reach unmanageable sizes. This results in the same problem as for web graphs were a need for compression arises as the graphs representing the problems grow too large to be held in main memory. The need for compression of graphs and adaptation of algorithms to run on these graphs are therefore growing ever more important.

1.1 Compression Methods

A lot of research has been done in the area of graph compression with existing methods including Virtual Node Miner [5], Dense Subgraphs with Virtual Nodes(DSM) [6], $k^2$-tree [10], WebGraph [2], Re-pair [9], a modified version of Re-pair by Claude and Navarro [4] and Graph Compression by BFS by Alberto Apostolico and Guido Drovandi [1]. Out of these $k^2$-tree, Re-pair and BFS are the only ones not specifically developed for use on web graphs. The compression methods that have been developed for use on web graphs are based on characteristics of these and therefore might not work as well on other types of graphs.

1.2 Problem Statement

Currently a lot of research has been focused on developing graph compression methods and evaluating them in terms of compression ratio and neighbour extraction times. There exists little research focused on comparing the search times on compressed graphs to the uncompressed versions. Even if the problem of available memory is solved through compression time is always an issue and being able to perform fast searches remain a problem. This thesis will therefore study whether graph compression on general graphs can result in better run times for search algorithms compared to the uncompressed graphs. This is done using four different compression methods and comparing the run times of the breadth first search (BFS) and depth first search (DFS) algorithms on
compressed and uncompressed graphs. The compression algorithms used for this purpose were chosen as they represent different methods of compression to get a more general representation of the properties of graph compression. These compression methods are RPGraph \cite{8}, LZGraph \cite{8}, DSM \cite{7} and \( k^2 \)-tree \cite{10}. The graph size range from 1000 nodes to 11000 nodes with densities ranging from 0.3 to 0.9.

2 Background

Compression algorithms make use of specific characteristics that different types of graphs can exhibit to achieve compression. These characteristics depend on the type of problem that the graph models. Certain graphs such as web graphs have areas containing dense sub graphs caused by the cross-linking within domains and between domains with similar purposes. Several algorithms developed for use on web graphs including DSM use this characteristic.

2.1 Dense Subgraph Miner

Dense Subgraph Miner (DSM) \cite{7} is a compression method originally developed for use on web graphs and uses the inherited characteristic of clustering in web graphs stemming from the fact that groups of web pages tend to link to each other \cite{8}. An example of this is web pages under the same domain linking to each other by the means of a menu. DSM achieves compression by finding these dense clusters of nodes and edges and replacing the edges between these with a virtual node and a fewer number of edges. Thus adding a small number of extra nodes but reducing the number of edges by a greater amount, as shown in figure 1. This process is then iterated until a sufficient compression has been reached or until each iteration no longer results in any significant compression.
2.2 $k^2$-tree

Graphs can be sparse or have sparse areas and this can be used to compress through smart data structures like $k^2$-tree \([10]\). A graph with any number of nodes and a density of 0.5 would have an adjacency matrix containing about 50% zeroes. $k^2$-tree eliminate large n by n areas of zeroes in the matrix by representing them by a single zero instead. An example of this is shown in figure 2 with the k-value set to two which means that in each iteration the
sub-matrices starting with the whole matrix, is each split in four \((2^2)\) equal sub-matrices. In the first iteration the roots’ child-nodes are generated with a self-value of 1 if and only if one node has an edge to another in the sub-matrix otherwise it is given a value of 0 and the iterative process stops for that branch. This process is then repeated until every branch ends in a zero or it has reached a one by one square. In figure 2 the original matrix has by this process been reduced from 16 integers to 12 which is a compression by 25% and on a sparser graph this number would be greater.

![Figure 2: k^2-tree](image)

### 2.3 Grammar-based Compression

Grammar-based compression methods such as Re-pair [9] and Lempel-Ziv use the fact that parts of the adjacency list for different nodes can be identical. Compression can be achieved by different means where the general idea is that the algorithm builds a dictionary by finding a number of similar sequences and replacing them with a common symbol. This process is then repeated until no similar sequences can be found or a certain compression has been achieved.

#### 2.3.1 RPGraph

RPGraph is an approximation of the Re-pair compression method and works as following [4]. The algorithm begins by traversing the adjacency lists counting the frequencies of every node pair. After this it retrieves the k most frequent pairs where k is a constant specified by the user. Then every occurrence of the node pairs is replaced by a new symbol. The last step is to restart the process of gathering pair frequencies where the last symbol was inserted, if the last adjacency list is reached it keeps going with the first list. This process is repeated until a certain number of iterations has been reached or no pair exists more than once.
2.3.2 LZGraph

LZGraph is an approximation of the Lempel-Ziv compression algorithm and works as following [4]. Lempel-Ziv is a method for compressing texts and LZ-Graph adapts this for use on graphs by interpreting every adjacency list as a string. The process begins by creating a dictionary of known phrases containing initially the empty string. The next step is to find the longest prefix of the text yet to be processed, which matches an existing phrase without crossing a list boundary. It then adds a new phrase to the dictionary with an identifier and the next character after the phrase. This is repeated until the end of the text.

2.4 Splitting a Graph

A common way to store a graph is in two parts; a list of the distinct labels of the nodes and a graph where the index of each name is used to represent it as a node in the graph. This is done to preserve space as an integer is a lot smaller than a string and since you only need the unique identifiers of the nodes to search the graph only the graph has to be loaded into main memory. Given a number of indexes as a result from the search it is easy to look up the string they correspond to therefore the whole process is not slowed down significantly by having to find the string afterwards. The same is also true if given a string; to find it’s index and run a search on the graph with it.
3 Method

3.1 Hardware

The machine used in our tests is a 2.66GHz Intel® Core™ i7-920 Processor (4 cores) with 16GB RAM, and a 1TB Disk (SATA 7200rpm) which runs 64-bit Ubuntu 14.04.

3.2 Compression Methods

To establish if compression results in better search times for graph algorithms than on the uncompressed graph we ran test with two graph algorithms. These were run on compressed and uncompressed graphs to be able to compare run times. Since different compression algorithms use different methods to compress graphs the different graph algorithms might perform better or worse as a result. Several compression methods were therefore used for each graph algorithm to get a more representative result for compression in general. The compression algorithms used in this paper are RPGraph and LZGraph by Francisco Claude and Gonzalo Navarro[8], DSM by Cecilia Hernández and Gonzalo Navarro[7] and $k^2$-tree by Susana Ladra[10].

3.3 Search Algorithms

To test the search times on the graphs compressed by the aforementioned methods two search algorithms were chosen. The algorithms are breadth first search (BFS) and depth first search (DFS), adapted for compressed graphs. These were chosen because they’re a part of several other algorithms for solving graph problems and therefore better speed in solving these would correlate to the same for the algorithms they are a part of.

The same implementation of BFS and DFS was used in the tests for all the different graphs, compressed and uncompressed. The only difference is how the graph is loaded into main memory and how neighbour lists were retrieved. The compression methods supplied their own functions for loading the graph from file and for extracting the neighbours of a node. The uncompressed graphs were represented by an adjacency list. The time it takes to load the graphs were not included in the measured times as these are not relevant to the thesis. DSM however had no functions for loading or extracting, instead the neighbour lists were loaded into integer arrays. The virtual nodes were identified by numbers greater than the number of nodes in the uncompressed graph.
3.4 Test Data

For the tests a large number of graphs of varying sizes and densities were generated with the random graph generator module from GTgraph\[^{11}\]. The random graph generator allows the user to specify the number of nodes and edges in the graph and then generates the graph by randomly adding edges between pairs of nodes. The smallest graphs generated had 1000 nodes, the largest had 11000 nodes and were generated in increments of 2000. For each size, graphs of densities 0.3, 0.5, 0.7 and 0.9 were generated where the density $D$ of a directed graph with no loops, $E$ edges and $V$ nodes is calculated by:

$$D = \frac{|E|}{(\left|V\right|(|V| - 1))}$$

For each of these densities five graphs were generated. Initial tests showed that increasing the number of graphs above this yielded marginally better precision.

To get representative test results 80 node-pairs were randomly generated as start and end nodes, for each test using BFS and DFS. This number was chosen by randomly generating node-pairs for a large graph and running the search algorithms until the cumulative average search time ceased changing by more than a microsecond. The graph we used for this test was a mid-dense graph with 10 000 nodes and 49 995 000 edges. This graph size is the approximate upper limit for the machine used for testing.

3.5 Measuring Memory Usage

The main memory usage for storing the graphs was measured as the search algorithms were running and compared to the usage of the uncompressed graph. This data was taken directly from the process information in Ubuntu’s system monitor.

3.6 Specific Settings

The graphs generated by GTgraph are not in the same format as required by the compression algorithms. Therefore these graphs had to be converted to a binary format specified by $<$nodes$>$ $<$edges$>$ -$<$node$>$ $<$sorted neighbour list$>$ -$<$node$>$ $<$sorted neighbour list$>$ where $<$nodes$>$ is an integer, $<$edges$>$ is an double and both $<$node$>$ and the neighbour list is integers.
3.6.1 Compression Method Settings

$k^2$-tree compression was achieved with

```
./build_tree <GRAPH><name><K1><K2> [<max level K1>] <S>
```

where the parameters set to 4 2 5 18 respectively on recommendation from Susana Ladra and then ./compress_leaves <name>. Graph compression by RPGraph was done by running ./compressGraph.py <GRAPH><mem><K> with <K> set to 100000 on recommendation and <mem> set to the size of the graph + 10%. DSM took the parameters <GRAPH><1> <1> <iterations> <subgraphsize> <outfile> where <subgraphsize> was set to 54. <iterations> were set to 5 for graphs with 5000 nodes or less, then 3 for 7000 or less and lastly only 1 iteration for those larger than that. Lastly LZGraph compression was simply done by running ./compressor <GRAPH>.

4 Results

4.1 Average Search Times

![Figure 3: average run time on graphs compressed with different algorithms](image)

The average run time for uncompressed graphs of varying sizes was calculated over densities ranging from 0.3 to 0.9 and for both BFS and DFS. Comparing run
times on the uncompressed graphs and graphs compressed by different methods shows that except for DSM, the compressed graphs greatly exceed the search times of the uncompressed graphs as shown in figure 3. RPGraph results in the fastest growing search time among the compression algorithms. The fastest search times for compressed graphs are achieved by DSM, appearing to be just as fast as the uncompressed graph, followed by LZGraph and then $k^2$-tree.

![Graph showing search times comparison](image)

**Figure 4:** average run time for BFS and DFS on uncompressed graphs and graphs compressed with DSM

Taking a closer look at how DSM performs shows that it does marginally worse than searches on the uncompressed graphs, with both having a close to linear increase in search time that is directly related to the size of the graph as seen in figure 4.
4.2 Average Search Times as Related to Density

Figure 5: run times for BFS on different densities
Figure 5 and 6 show the average run time for different compressions and graph densities. Since the time for DSM and the uncompressed graphs are so much faster than the other compressions it does not show in the diagram. For the results on DSM and the uncompressed graphs see figure 7 and 8.

Figure 5 and 6 further show that the search times on uncompressed graphs eclipse the compressed ones. RPGraph has the slowest run times which peak at density 0.7. $k^2$-tree has better search times and they get better as the graphs density increases. LZGraph has overall better search times than both and has the same characteristic as RPGraph. DSM has the best search times on compressed graphs and the time does not vary much with density.
Figure 7: run times for BFS on graphs compressed with DSM and on uncompressed graphs with different densities

The search times with BFS on graphs compressed by DSM and on the uncompressed show little change for different densities.
Figure 8: run times for DFS on graphs compressed with DSM and on uncompressed graphs with different densities

The same stands true for DFS with little change for different densities.
The average compression ratios calculated from the main memory usage shows that DSM achieves minuscule compression on general graphs. The other algorithms achieve significant compression.

5 Discussion

5.1 Method Discussion

As mentioned in the method precautions were made to ensure the correctness of the results. Both the number of graphs per density and number of node pairs used as start and end for the search algorithms were determined through initial testing. Running the tests with more graphs and running a total search would give more precise results, but the initial tests showed that the numbers used give a close approximation.

The graph sizes were not determined initially. It was an iterative process where graphs were generated as needed so that if the trend remained unchanged larger leaps in size could be made and points of change could be focused on. No such
point was ever reached since the trend showed a continuous growth. More data points would have made the results more precise but would likely not have affected the trend of growth.

All tests were run on a single machine to allow the run times to be compared accurately. Running on several machines could have ensured that the trend was consistent over several environments. This however would have been too time consuming and would produce minimal useful results.

In the tests four compression methods were used. There exists other compression methods which might perform better in regard to search times. Each of the four methods compress through different means and therefore should give a general idea of how compression affects search times.

Two different graph search algorithms were used in the tests. All graph algorithms have to traverse the graph in some way and if the time to retrieve neighbours is greater for compressed than uncompressed graphs similar results would be expected for any additional algorithms.

5.2 Discussion of Results

The results show that the search algorithms are faster on uncompressed graphs than on compressed graphs when both can be held in main memory and this holds true for every size and density. This is most likely due to additional overhead causing longer retrieval times on the compressed graphs whilst the neighbours in the uncompressed graphs can be directly accessed through the adjacency list. The large difference could be a consequence of the compression algorithms aiming to compress the graphs as much as possible and focusing less on retrieval times. Therefore it might be possible to decrease search times on the compressed graphs by sacrificing some compression in exchange. This could be what happens when compressing with DSM as shown by its low search times in figure 3 and how low its compression ratio is in figure 9. The reason for the low compression ratio might be because DSM was developed for use on web graphs and look for dense subgraphs which the random generated graphs used in the tests appear to not have many of. Even if the graph itself is dense this only means that there are many edges per node but not that there are many dense clusters.

The results further more show that there is no significant difference between the search times of BFS and DFS as related to the different densities. Search times on graphs compressed by $k^2$-tree decrease as density increases as shown in figure 5 and 6. RPGraph and LZGraph on the other hand show a tendency to have worse search times on mid-dense graphs than at lower or higher densities. DSM followed a similar pattern to the uncompressed graph with a linear increase in search times directly related to the number of nodes. This coupled with the fact that DSM has poor compression rates on general graphs means that it is not suitable for searching general graphs. Since LZGraph compression appears
to have the fastest neighbour queries of the remaining compression algorithms considered in our tests it might be preferable for general graph compression when the graph can not fit in main memory. If we were to study larger graphs that could only fit in main memory compressed the slow neighbour queries might outweigh the slow disk access times required to search the uncompressed graphs.

5.3 Future Research

The scope of this thesis limited the tests to uncompressed graphs that can fit in main memory. A future topic for research can therefore be to study the effects on search times that compression has compared to uncompressed graphs that can not be held in main memory. For this purpose certain compression methods such as DSM and RPGraph can be excluded in favour of those who performed better. Another area of study could be to see if compression algorithms could be modified to improve neighbour query time.

6 Conclusion

Search algorithms are faster on uncompressed graphs than on graphs compressed by the compressions algorithms tested, if both versions fit in main memory. When looking at how the search algorithms perform on graphs of specific densities the same is also true. At no density does the search algorithm perform better on the compressed graphs than on the uncompressed.
References


