Computers, Materials & Continua CMC, vol.63, no.2, pp.743-753, 2020

Expanding Hot Code Path for Data Cleaning on Software Graph

Guang Sun[1, 2, *], Xiaoping Fan[1], Wangdong Jiang[1], Hangjun Zhou[1], Fenghua Li[1] and Rong Yang[1]

Abstract: Graph analysis can be done at scale by using Spark GraphX which loading data into memory and running graph analysis in parallel. In this way, we should take data out of graph databases and put it into memory. Considering the limitation of memory size, the premise of accelerating graph analytical process reduces the graph data to a suitable size without too much loss of similarity to the original graph. This paper presents our method of data cleaning on the software graph. We use SEQUITUR data compression algorithm to find out hot code path and store it as a whole paths directed acyclic graph. Hot code path is inherent regularity of a program. About 10 to 200 hot code path account for 40%-99% of a program's execution cost. These hot paths are acyclic contribute more than 0.1%-1.0% of some execution metric. We expand hot code path to a suitable size which is good for runtime and keeps similarity to the original graph.

Keywords: Hot code path, expanded hot path, software graph, software graph.

1 Introduction

According to the definition from Wikipedia, Big data is a term used to refer to datasets that are very large or complex for traditional data-processing application software to adequately deal with. Big data was originally associated with a large volume of data (volume), rapid data generation (velocity), a variety of data types (variety), and uncertainty of data source and derived data (veracity) [Li, Li, Chen et al. (2018)].

There are five steps to deal with big data. Collecting data, cleaning data, storing data, analyzing data and data's further analyzing [Deelman, Peterka, Altintas et al. (2017)]. Collecting data is that people use tools like web crawler to collect data from nature or the Internet. Due to some of this data is useless such as empty data, the second step works for cleaning data [Djidjev, Chapuis, Andonov et al. (2015)]. After cleaning, we need to store the cleaned data into massive storage or distribution system. Analyzing data aims to use some techniques for dealing with cleaned data like Hadoop. The last step is further analyzing. If the result of the handled data is needed to keep using by some equipment such as The Google driverless system, further analyzing will be carried out in the next process [Zhang and Xiong (2016)]. If the results will be read by a human. The best way is visualization because of its immediacy, convenience [Andreas, Nicolai and Andreas

[1] Institute of Big Data, Hunan University of Finance and Economics, Changsha, 410205, China.

[2] School of Engineering, University of Alabama, Tuscaloosa, 35487, USA.

[*] Corresponding Author: Guang Sun. Email: simon5115@163.com.

Received: 01 January 2019; Accepted: 12 April 2019.

(2016); Dijkman, Dumas, Dongen et al. (2011); Fiannaca, Rosa, Rizzo et al. (2014)].

Data cleaning is the second process of detecting, correcting and removing useless or inaccurate data from a recordset, and transforming data from one "raw" data into another appropriate format, making it more suitable and valuable for the downstream purposes such as analytics [Sohangir, Wang, Pomeranets et al. (2018); Nwagwu, Okereke and Nwobodo (2017); Tan, Blake, Saleh et al. (2013); Iqbal, Luo, Khan et al. (2018)]. Graphs have been extensively used in real-world applications. In big data platforms, graph analysis can be done at scale by using Spark GraphX which loading data into memory and running graph analysis in parallel. In this way, we should take data out of graph databases and put it into memory [Koohi and Zahedi (2017); Schoknecht, Thaler, Fettke et al. (2017)]. Considering the limitation of memory size, the premise of accelerating graph analytical process reduces the graph data to a suitable size without too much loss of similarity to the original graph [Maguire, Rocca-Serra, Sansone et al. (2013); Bollegala, Weir and Carroll (2013)].

This paper focuses on data cleaning on the software graph. We show how to use the SEQUITUR data compression algorithm to find out hot code path and store it as a whole path directed acyclic graph. Hot code path is inherent regularity of a program. About 10 to 200 hot code path account for 40-99% of a program's execution cost. These hot paths are acyclic contribute more than 0.1-1.0% of some execution metric. We expand hot code path to a suitable size which is good for runtime and similarity to the original graph.

2 SEQUITUR data compression algorithm

SEQUITUR data compression algorithm is used to capture the inherent regularity of the application execution and builds the representation of the regularity in the compressed format. The algorithm operates in linear time and space, shown as follow [Zeng and Church (2009); Tarjan (1972); Bentley and Sleator (1986); Myles and Collberg (2004)].

Before running this algorithm, we should execute the application with a given input set, and record the sequence of code blocks. A code block is a set of statements, signal entry, signal exit, no loop and branch statements, and considered as a token in the algorithm. SEQUITUR algorithm works by scanning a sequence of tokens and building a list of all the token pairs. Whenever the same occurrence of a pair is discovered, the two occurrences are replaced in the sequence by an invented non-terminal token, the list of token pairs is adjusted to match the new sequence, and scanning continues. If a pair's non-terminal token is used only in the created token's definition, the token is replaced by its definition and the token is removed from the defined non-terminal tokens. When the scanning process completes, the sequence can be interpreted as a compressed context-free grammar that infers a hierarchical structure which can be stored as a whole path directed acyclic graph (DAG) [Myles and Collberg (2004)]. The rule definitions for the non-terminal tokens can be found in the list of token pairs.

The algorithm appends each token to the end of rule T and searching the resulting rule and all other production rules for redundancy. If the tail of the new end of the compressed trace matches any of the production rules, produces an appropriate substitution. Then, if any redundancy is found a new production rule is added and substituted in the compressed trace. The algorithm also checks each time a replacement is made to ensure

that each production rule is used more than once. If any rule is used just once its production is substituted where it is used and the rule is removed from the grammar. This prevents the retention of rules that do not contribute to compression.

```
algorithm SEQUITUR( S )
input Execution Trace S
output Grammar G
Grammar G
Rule T
1. for each token in S
2. append token to end of production for T
3. if duplicate digram appears
4. if other occurrence is a rule g in G
5. replace new digram with non-terminal of g.
6. else
7. form a new rule and replace duplicate
8. digrams with the new non-terminal.
9. if any rule in G is used only once
10. remove the rule by substituting the production.
11. return G
```

Figure 1: SEQUITUR data compression algorithm

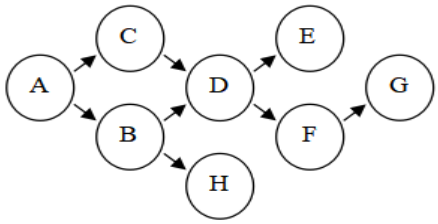

Figure 2: Example of block call graph

Our method requires the program to be instrumented at block's entry and return. This means the sequence contains block's name, return r, and program exits x. Sometimes, one block may have more than one r, this means the block is called multiple times. Here it is an example to explain how SEQUITUR algorithm works. The call blocks are shown in Fig. 2, and the accordingly sequence is:

A B H r r C D E r r r r x A B H r r r x A B D F r r r r x

Initially, the rule T is empty.

T→

Algorithm starts on A, travels the execution trace, appends A B H to the right side of T.

T→A B H

H is the end of a runtime, the algorithm travels back to the A and adds two r to the T.

T→A B H r r

Travels another trace, we get C D E.

T→A B H r r C D E

Likes H, E is the end of this runtime, the algorithm travels back to the A, if E was called two times, the algorithm should append two r, so finally adds four r and one x to the T.

T→A B H r r C D E r r r r x

Each step, the algorithm checks the last two tokens. If the string is duplicated, the algorithm constructs a rule and applies it to T. In this case rr appears twice, so

T→A B H 1 C D E 1 1 x

1→r r

The algorithm continues appending tokens and seeking duplication. In this case another rr is found, replace it with symbol 1. When there are no more tokens to process for first trace, the result is:

T→A B H 1 C D E 1 1 x

1→r r

Continue running this algorithm, we can get the following sequence.

T→A B H 1 C D E 1 1 x A B

1→r r

T→A B H 1 C D E 1 1 x A B

1→r r

2→A B

T→2 H 1 C D E 1 1 x 2

1→r r

2→A B

T→2 H 1 C D E 1 1 x 2 H

1→r r

2→A B

T→3 1 C D E 1 1 x 3

1→r r

2→A B

3→2 H

T→3 1 C D E 1 1 x 3 1

1→r r

2→A B

3→2 H

4→3 1

T→4 C D E 1 1 x 4 x

T→4 C D E 1 1 x 4 x 2 D F G 1 1 x

1→r r

2→A B

3→2 H

4→3 1

5→1 1
6→5 x

The final result is:
T→4 C D E 6 4 r x 2 D F 6
1→r r
2→A B
3→2 H
4→3 1
5→1 1
6→5 x

Its representation is a whole path DAG as shown in Fig. 3.
SEQUITUR is an online algorithm, runs in time O(N), where N is the trace length. The size of the compressed trace is O(N), worst case (no compression possible) and O (logN) best case.

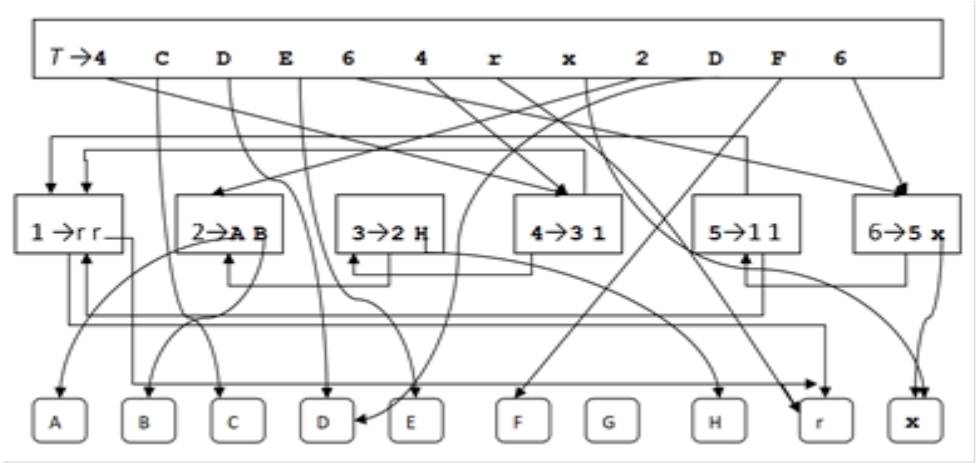

Figure 3: whole path DAG

3 Hot code path expansion

A hot code path refers to code sequence executed most often. Ammons, Ball, and Larus showed that few relatively hot code paths, about 10 to 200 account for 40-99% of a program's execution cost. These hot paths are acyclic contribute more than 0.1-1.0% of some execution metric [Myles and Collberg (2004)].

```
HotPathExpand( G, HP )
input Whole Path DAG G
input Hot path HP
output Expand Set, I
Set I=set of expanded assemblies of HP, initially empty.
TempI=set of tokens, initially empty.
Queue S=queue of temporary up tokens; initially empty.
Tokens t=empty.
Integer retu_sr, retu_pr,succ_num=0.
add each node hpc in HP to I and queue S
while S is not empty
{t=head of S
for each node p whose production contains t
{if p=T
then add t to TempI
else
add tokens appear in the left of p to S
}
delete head from S
}
for each trace Tr contains c in TempI
{
for each successor sr of c
 { if (sr is terminal node )
then if (sr==r )
then retu_sr++
else add sr to I
else
travel sr and add each terminal node to I }
for each predecessor pr of c
{if (pr is terminal node)
then if (pr==r)
then retu_pr++
else if (retu_pr==0)
then add pr to I
else retu_pr--
else
{travel pr and add each unreturned terminal node to stack TempS
while (retu_pr>0)
{
pop(TempS)
retu_pr-- }
add TempS to I} } } }
return I
```

Figure 4: The hot code path expansion algorithm

Larus's algorithm interested in searching for minimal (shortest) hot code paths. This means the hot code paths most likely are of 100-1,000 acyclic paths long. This is not

enough to represent the whole program. Even the longer hot code neither. If we try to get an appropriate software graph stands for the whole big graph. We should expand the hot code path to a suitable size.

The hot code path expansion algorithm we use in this paper is based on software change impact analysis and shown in Fig. 4. Our algorithm collects every dynamic path that connects the hot code path. Every block which is called after the hot path, and any block which is on the call stack after any block of hot path returns, is moved in the set of expanding.

Our code path expansion algorithm starts on a given hot code path HP, the block hpc is the first one. Our algorithm ascends upward hpc in the given DAG, each execution trace Tr which contains hpc will be found out. Recursively the algorithm searches forward for blocks that are called directly or indirectly after hpc. By searching backward in the execution trace the algorithm finds out the blocks return to hpc.

For example, if we expand the DAG shown in Fig. 4, the hot code path is {A, B, H}. We start the process from node H.

Step 1. Ascending upwards to the node 3→2 H. Because this node is not T, continue ascending upwards.

Step 2. Ascending upwards to the node 4→3 1 which contains rule 3. Because this is not T, continue ascending upwards.

Step 3. Ascending upwards to T, 4 appears twice, so we find out two execution traces.

Step 4. Traveling trace "4 C D E 6", adds C, D, E to set I. Nodes C, D, E are called directly or indirectly after H.

Step 5. 6 is a non-terminal node, search the rules recursively and replace it with terminal node r, r, r, r, x. x stands for the end of this execution trace. Variable retu_sr records the number of returns after token 4. When the algorithm meets x, the process of searching forward ends, the value of retu_sr is 4, there are our times returning happened in the execution trace.

Step 6. Return C, D, E, and set retu_sr to 1. 4 is a non-terminal node, before searching backward that will result in returning to H, travel it recursively and replace 4 with terminal nodes A, B, H, r, r.

Step 7. Adds A, B to set I. There is no token r before H, that is no block returns before H, the value of retu_pr is 0.

Step 8. If the value of retu_pr is greater than 0, uses stack TempS to skip the blocks which already returned. Traveling execution trace "4 r x", get A and B. Because A and B are in set I, skip them. So, the expanding of H is {A, B, C, D, E}.

Step 9. Running the algorithm to block A, ascends upward and meets 2 and 4. Traveling 2 in T, gets the expanding set {B, D, F}.

Step 10. Traveling 4 in T, gets the expanding set {A, B, C, D, E}

Step 11. Merging the sets, we can see the expanding set of A is {A, B, C, D, E, F}.

Step 12. Input block B to the algorithm, output the expanding set {D, F}.

Step 13. Merge the sets of A, B, H, output the expanding set {A, B, C, D, E, F} of the hot code path {A, B, H}. The hot code path expansion algorithm runs in time O (P2), P is the number of nodes.

4 Experiment and visualization analysis

Let p' be the hot code path obtained from program P, p'' be the hot code path expansion from p'. Ammons and Larus defined hot code paths p' as acyclic paths that contribute 0.1-1.0% of some execution metric. In the SPEC benchmarks, they showed that relatively few p' (10-200) account for 40-99% P execution cost. Usually, p' is 100-1,000 acyclic paths long. There is no limitation for p'', that is theoretically speaking, a hot code path can be expanded over every block. But using our algorithm, the expanding process will end in the x. This makes sure the expanding process runs in P execution trace, and end in the execution trace as well. It's impossible to let the algorithm expand to every block.

We randomly download 1200 software applications from sourceforge.net, and abandon 167 applications of them which user ratings stars less than 2. For the rest applications, calculating the hot code path and expanding them. When we look at the size comparison between $p'$and p'' show in Fig. 5, we can know that most size of p'' is four time that of p', 400-4,000 acyclic paths long.

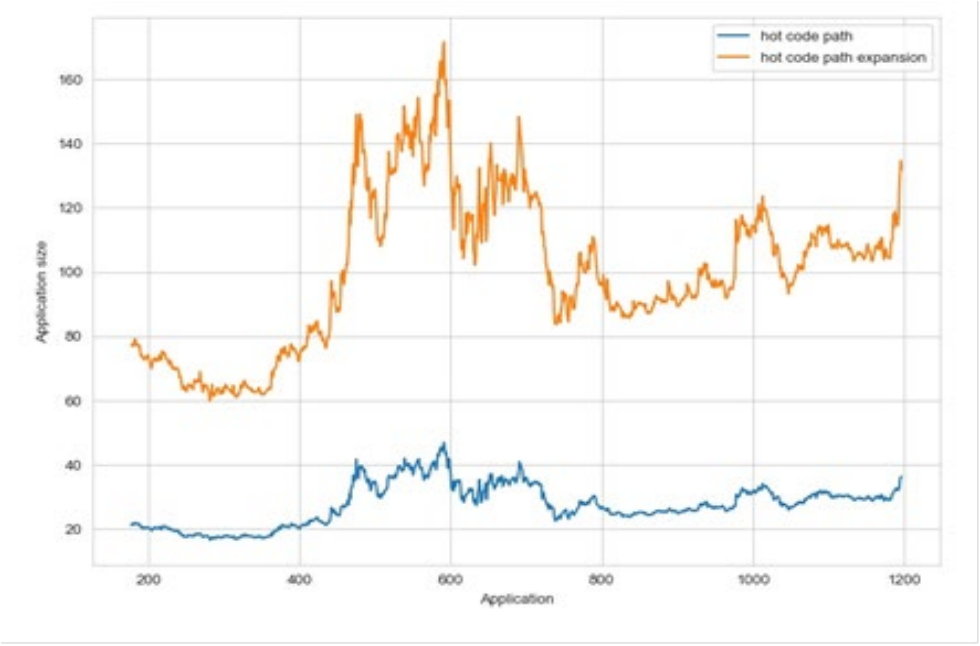

Figure 5: Size comparison between hot code path and expanded hot code path

We use resemblance which is come from software birthmarking to measure the similarity of two programs. We define the resemblance of P and p':

$$R(p,p') = \frac{|p \cap p'|}{|p \cup p'|} \tag{1}$$

Likewise,

$$R(p,p'') = \frac{|p \cap p''|}{|p \cup p''|} \tag{2}$$

Here the $\cap$ operation is common subgraph, $\cup$ is subgraph responses for 95% runtime cost. Because the core component may be the hot code path, we use the resemblance to

accurately measure the substitutability between P and p', P and p''. From Fig. 6, we can see that the substitutability between P and p'' is much higher than P and p'.

Figure 6: The comparison with the substitutability

5 Conclusion

Graphs provide a better way of dealing with software applications like assembly call, control flow and data flow. Often, people model analytics software applications in the form of DAGs [Schrijver (2012)]. Thus, Graph Databases have become alternatives to SQL and NoSQL databases and common computational tools. Analysis and processing of very large graph data sets poses a significant challenge. Extracting valuable software information from graph big data requires efficiently process. Common programming languages are used as the interface with graph analysis and often embedding it into standard applications. In this situation, reducing graph big data to a suitable scale is the effective method [Moon, Lee, Kang et al. (2016)].

Hot code paths make an essential part of the program execution. It is a good alternative to whole software application graph. But usually, hot code paths are 100-1,000 acyclic paths long. For graph analysis, the similarity to the original graph is low. In order to clean software graph data. We use SEQUITUR data compression algorithm to find out hot code path and store it as a whole path directed acyclic graph. Then, we expand hot code path to a suitable size which is good for runtime and keeps similarity to the original graph. We randomly download 1200 software applications from sourceforge.net, and calculate the hot code path, expand them. Most expanded hot code paths are of size four time that of the original graph, 400-4,000 acyclic paths long. Substitutability between expanded hot code paths and original graphs is much higher than original graphs and hot code paths.

Expanded hot code paths are more suitable for efficiently big data graph process.

Acknowledgement: This research work is supported by Hunan Provincial Education Science 13th Five-Year Plan (Grant No. XJK016BXX001), Social Science Foundation of Hunan Province (Grant No. 17YBA049), Hunan Provincial Natural Science Foundation of China (Grant No. 2017JJ2016). The work is also supported by Open foundation for University Innovation Platform from Hunan Province, China (Grand No. 16K013) and the 2011 Collaborative Innovation Center of Big Data for Financial and Economical Asset Development and Utility in Universities of Hunan Province. National Students Platform for Innovation and Entrepreneurship Training (Grand No. 201811532010). We also thank the anonymous reviewers for their valuable comments and insightful suggestions.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Andreas, S.; Nicolai, F.; Andreas, O. (2016): Process model search using latent semantic analysis. *International Conference on Business Process Management*, pp. 283-295.

Bentley, J. L.; Sleato, D. D. (1986): A locally adaptive data compression scheme. *Communications of the ACM*, vol. 29, no. 4, pp. 320-330.

Bollegala, D.; Weir, D.; Carroll, J. (2013): Cross-domain sentiment classification using a sentiment sensitive dictionary. *IEEE Transactions on Knowledge and Data Engineering*, vol. 25, no. 8, pp. 1719-1731.

Deelman, E.; Peterka, T.; Altintas, I.; Carothers, C. D.; Kerstin, K. V. D. et al. (2017): The future of scientific workflows. *International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 159-175.

Dijkman, R.; Dumas, M.; Dongen, B.; Kaarik, R. (2011): Similarity of business process models: metrics and evaluation. *Information Systems*, vol. 36, no. 2, pp. 498-516.

Djidjev, H.; Chapuis, G.; Andonov, R.; Thulasidasan, S.; Lavenier, D. (2015): All-pairs shortest path algorithms for planar graph for GPU-accelerated clusters. *Journal of Parallel and Distributed Computing*, vol. 85, no. 3, pp. 91-103.

Fiannaca, A.; Rosa, M. L.; Rizzo, R.; Urso, A.; Gaglio, S. (2014): An expert system hybrid architecture to support experiment management. *Expert Systems with Applications*, vol. 41, no. 4, pp. 1609-1621.

Iqbal, M. S.; Luo, B.; Khan, T.; Mehmood, R.; Sadiq, M. (2018): Heterogeneous transfer learning techniques for machine learning. *Iran Journal of Computer Science*, vol. 1, no. 1, pp. 31-46.

Koohi, T.; Zahedi, M. (2017)**:** Scientific workflow clustering based on motif discovery. *International Journal of Computer Science, Engineering and Information Technology*, vol. 7, no. 4, pp. 1-13.

Li, Y.; Li, J.; Chen, J.; Lu, M.; Li, C. (2018): Seed selection for data offloading based on social and interest graphs. *Computers, Materials & Continua*, vol. 57, no. 3, pp. 571-587.

Maguire, E.; Rocca-Serra, P.; Sansone, S. A.; Davies, J.; Chen, M. (2013): Visual

compression of workflow visualizations with automated detection of macro motifs. *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 12, pp. 2576-2585.

Moon, S.; Lee, J. G.; Kang, M.; Choy, M.; Lee, J. W. (2016): Parallel community detection on large graphs with MapReduce and GraphChi. *Data & Knowledge Engineering*, vol. 104, no. 1, pp. 17-31.

Myles, G.; Collberg, C. (2004): Detecting software theft via whole program path birthmarks. *ISC 2004: Information Security*, vol. 3225, pp. 404-415.

Nwagwu, H. C.; Okereke, G.; Nwobodo, C. (2017): Mining and visualising contradictory data. *Journal of Big Data*, vol. 4, no. 1, pp. 36-47.

Schoknecht, A.; Thaler, T.; Fettke, P.; Oberweis, A.; Laue, R. (2017): Similarity of business process models-a state-of-the-art analysis. *ACM Computing Surveys*, vol. 50, no. 4, pp. 52-85.

Schrijver, A. (2012): On the history of the shortest path problem. *Documenta Mathematica*, vol. 17, no. 1, pp. 155-167.

Sohangir, S.; Wang, D.; Pomeranets, A.; Khoshgoftaar, T. M. (2018): Big data: deep learning for financial sentiment analysis. *Journal of Big Data*, vol. 5, no. 1, pp. 3-28.

Tan, W.; Blake, M. B.; Saleh, I.; Dustdar, S. (2013): Social-network-sourced big data analytics. *IEEE Internet Computing*, vol. 7, no. 5, pp. 62-69.

Tarjan, R. (1972): Depth-first search and linear graph algorithms. *12th Annual Symposium on Switching and Automata Theory*, vol. 1, no. 2, pp. 146-160.

Zeng, W.; Church, R. L. (2009): Finding shortest paths on real road networks: the case for a*. *International Journal of Geographical Information Science*, vol. 23, no. 4, pp. 531-543.

Zhang, D.; Xiong, L. (2016): The research of dynamic shortest path based on cloud computing. *12th International Conference on Computational Intelligence and Security*, pp. 452-455.