Condensed Representation of Graph as Adjacency Matrix with parallel edges

Maharshi Roy, Rajat Kumar Sahu, Amrit Daimary

ism2014006@iiita.ac.in
ihm2014501@iiita.ac.in
irm2014001@iiita.ac.in

Abstract—*This paper gives an insight into the various methods that can be applied in condensation of an undirected graph represented as a 2D-matrix (Adjacency Matrix). Adjacency matrix representation is very efficient as one can retrieve connectivity information in a constant time O(1) operation. However, if the number of vertices is very large, and the connectivity is sparse, this representation can lead to a lot of space wastage, and also infeasible in terms of memory usage (O(v^2) space complexity). Thus we need some methods to condense the matrix without losing information (lossless). This paper discusses approaches that can be applied in practical scenarios. The paper considers the case of graphs having parallel/multiple edges between 2 vertices (a case ignored by most algorithms).*

Keywords — *Matrix Condensation, Parallel edges, Adjacency Matrix*

MOTIVATION

Compression has always been a demanding and debated topic in research groups, as in practical scenarios, we are bound by the availability of memory in any computation task. In a matrix compression involving the representation of a graph, one cannot lose connectivity information, thus allowing us to only perform lossless compression techniques. Performing operations on uncompressed matrix can lead to high memory consumption. Computing with large matrices means we need to access RAM more frequently, as we can't keep much data in the CPU registers. The average latency of a RAM read operation is 150 times that of a CPU register access.
The paper discusses three variations of bitstring compression, which allows large matrices to be representable by integer values that can be directly moved to a faster cache or CPU registers. Thus, not only matrix compression helps dealing with memory limitation, but it also indirectly handles the case of latency or time consumption.

METHODOLOGY AND APPROACH

As said in previous sections, we will delve into 3 variations of bitstring compression techniques, that can be appropriately chosen according to the scenario, and analyze their efficiency and fallacies. The approaches discussed here considers the undirected graph as having parallel edges between same pair of vertices. This leads to additional complications as the entries in the adjacency matrix is not just binary. However, the methods discussed here are versatile enough to handle such cases.

1) All the methods discussed are performed after doing trivial space reduction in case of undirected graphs. In an undirected graph, there is bidirectional connectivity between vertices.

$$v \rightarrow u \Leftrightarrow u \rightarrow v$$

Thus, we store lower vertex information in higher vertex.

```
if (v>u)
        v[u]=cnt;
else
        u[v]=cnt;
```

This leads us naturally to avoid storing duplicate information and store connectivity information only once. This leads to storing only upper triangle or lower triangle rather than the whole matrix. This reduces consumed space by 1/2 (a significant reduction).

2) To account for multiple edges between same pair of vertices, we store the count of such vertices as Aij..
Now, let's discuss the methods one by one:-

Integer conversion method using appropriate Base

Supposing we have the connectivity information of vertex 8 as:-

8: {0,2,5}

Instead of storing 3 separate integers, we can choose an appropriate base and convert it to a single integer to be stored at an array with index 8. In more generic terms, given:-

$$v_1 : \{u_1, u_2, u_3, \ldots\ldots, u_n\}$$

We can choose a base

$$b = max(u_1, u_2, u_{3,}......., u_n) + 1$$

and convert $\{u_1, u_2, u_{3,}......., u_n\}$ to:-

$$v_1 = u_1 + u_2 * b + u_3 * b^2 + u_n * b^{n-1}$$

Thus a matrix like:-

0					
1	0				
2	1	0			
3	2	0	0		
4	3	2	0	0	
5	2	1	0	0	0

gets converted to:-

0	
1	0
2	1
3	2
4	23
5	12

Thus the space complexity earlier was (considering complete graph):

$$1 + 2 + 3 + 4 + ...(n-1) = O(n*(n-1)/2) \approx O(n^2)$$

But now conversion to each integer is storing same information as a single integer:

$$1 + 1 + 1 + ... (n\ times) = O(n)$$

Algorithm:-

```
//Choosing appropriate Base
Base = 0
for i=1 to N
    for j=1 to i-1  //Iterate on all elements
        Base ← max(Base,graph_ij)
//Condensation
for i = 1 to N
    val ← 0
    for j = i-1 to 0
        val ← val * Base + graph_ij
    condensed_i ← val //A linear array
//Decondensation
for i = 1 to N
    val ← condensed_i
    for j = 1 to i-1
        decondensed_ij ← val % Base
        val ← val / Base
```

Drawbacks:

This method may seem lucrative, but has a problem with overflow. Integer (even unsigned long long int in 64-bit architecture) has a limited size (64 bit max in most systems). Thus most computers can at max store $2^{64} - 1$.
However, if the Base chosen is large and also the matrix is very dense and large, huge numbers can get generated that can lead to overflow.

Bit-string Method (Static length)

In bit string method, we still use 64-bit integers to store information. However, we don't waste space by keeping leading zeros for a small number. That way whole 64-bit gets utilized, and for sparse graphs we can keep edge information for multiple vertices in one single integer.
We do that by finding out the maximum bit length required to represent all the values stored in the matrix. Let's look at an example:-

Edge	Count	Binary Rep.	Bits required
1,1	900	1110000100	10
1,2	1023	1111111111	10
1,3	721	1011010001	10
1,4	256	100000000	9
1,5	1	1	1
1,6	10	1010	4
1,7	700	1010111100	10
1,8	20	10100	5

In this case, we would get away with 10 bits (max bit length) for each edge value.

We then divide a 64-bit integer into 10-bit chunks, and store each edge information sequentially:-

$bitstring_1$ = 0000 0000001010[10] 0000000001[1]

0100000000[256] 1011010001[721] 1111111111[1023]

1110000100[900]

Whenever we are unable to accommodate all the edges, we append another 64-bit integer:-

$bitstring_2$ = 0000 0000000000 0000000000

0000000000 0000000000 000010100[20] 1010111100[700]

We keep track of number of edges found till now and stop accordingly.
This method works in average case very well, because the whole matrix can be reduced to a list of 64-bit integers.

Algorithm:-

```
//find maximum bit length required
maxlen = 0
for i=1 to N
        for j = 1 to i-1
        maxlen ← max (maxlen,
length_binary(graph_ij))
//length_binary is an utility function that
returns length of binary string. Implementation
is trivial.
//create bit string integers and append to list
ordered_list = {}
offset ← 0
for i = 1 to N
        for j = 1 to i-1
        val ← graph_ij
        if (offset + maxlen ≥ 64)
                add old integer to list
                create new integer x
                offset ← 0
        val ← val << offset //Right shift by
offset
        offset ← offset + maxlen
        x ← x  | val  //logical OR
```

Drawbacks:

For an immensely dense graph, where all matrix values are of order ~ $2^{(50-60)}$, almost a single integer is required for all the edges. In such a worst case, space required is more or less same. However, such dense graphs rarely arise, and we can get away with the excellent average case compression rate.

<u>Bit-string Method (Variable length)</u>

The method is almost similar to the above one however, we can reduce bit-space wastage further by noticing that many small edge values will have leading zeros in case max bit length is large. As in the above example, max bit length was 10, however,to represent 1, we require only 1 bit, rest of the 9 bits account for leading zeros and bit-space wastage. Similar for value 10, we could do with 4 bits only.
In variable length method we keep 2 separate set of integers:-

length-string: stores the number of bits required by M_{ij}
value-string: stores the value M_{ij}

However, in the length string, we need to store each length value in a set of maximum bits required to store the lengths. In this case, we have maximum length value: 10, for which we require 4 bits. Thus we would store length values in intervals of 4.
As an example, we do as:-

length-string: 0101[5] 1010[10] 0100[4] 0001[1] 1001[9] 1010[10] 1010[10] 1010[10]

bit-string: 10100[20] 1010111100[700] 1010[10] 1[1] 100000000[256] 1011010001[721] 1111111111[1023] 1110000100[900]

Hence, we can see that we accommodated all the edges in a single bit-string, with a tradeoff of keeping a length-string integer. However, this tradeoff is acceptable as the space saved by truncating leading zeros in edge-string is immense.
Algorithm:-

```
length-string int L
edge-string int E
offset l ← 0  //offset for length-string
offset e ← 0  //offset for edge-strings
for i = 1 to N
for j = 1 to i-1
        val ← graph_ij
        bitlen←length_binary(graph_ij)
        if (e + bitlen ≥ 64)
                append old E to list
                allocate new E
                e ← 0
        if (l + maxbits ≥ 64)
                append old L to list
                allocate new L
        l ← 0
val ← val << e
E ← E | val
e ← e + bitlen
bitlen ← bitlen << l
L ← L | bitlen
l ← l + maxbits
```

Drawbacks:

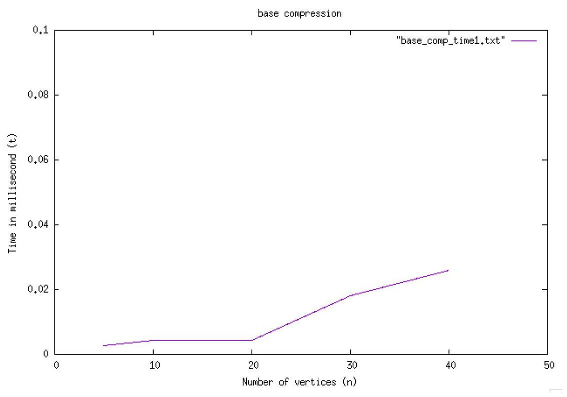
It also has the same drawback as its static counterpart. i.e. for large matrix values, almost a single integer is required for all the edges. In such a worst case, space required is more or less same.

IMPLEMENTATION & RESULTS

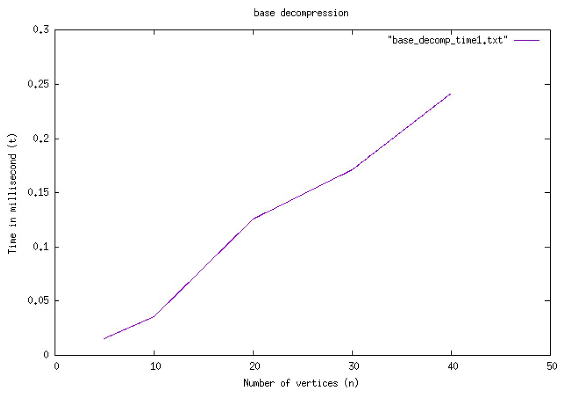
All the 3 algorithms were implemented in C++ and Time and Space complexity analysis were performed on all the three methods, and line-graphs were obtained for varying inputs. All these operations were done in GNUPlot. All the tasks was performed in a 64-bit machine (x86-64 architecture). The line-graphs for the various methods are given as follows:-

1) Base Conversion Method

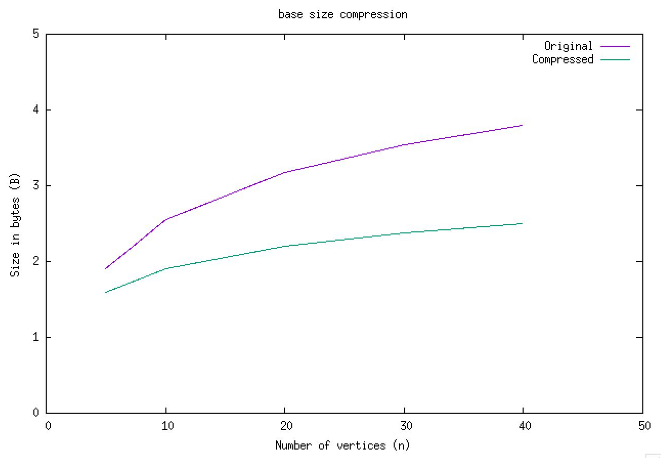
a) Compression Time (Time complexity)



b) Decompression Time (Time complexity)

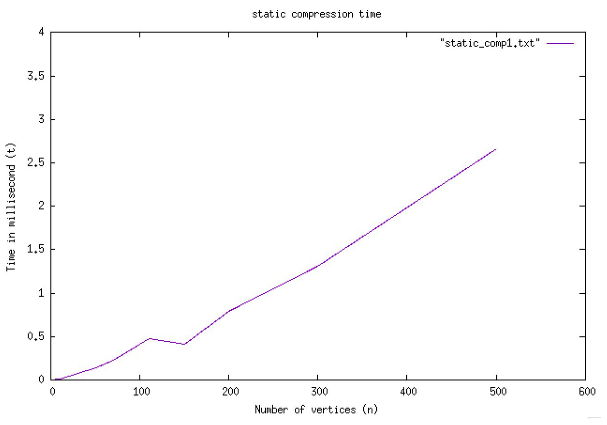


c) Size comparison (Space complexity) [Log scale]

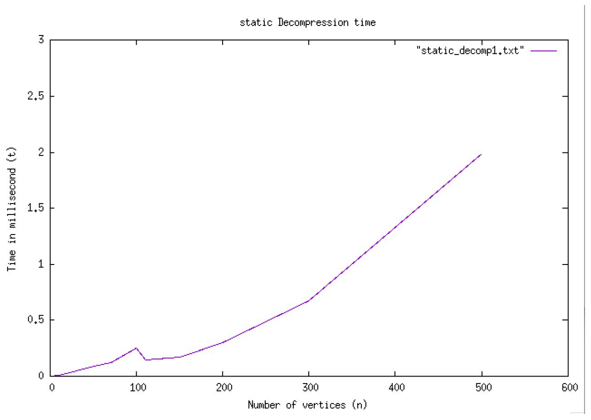


2) Bit-string method (Static length)

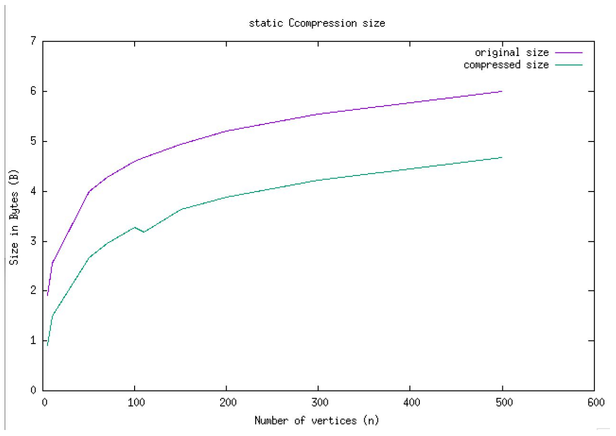
a) Compression Time (Time complexity)



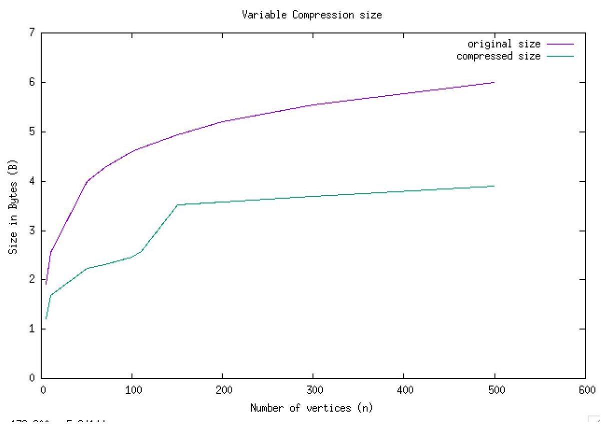
b) Decompression Time (Time complexity)



c) Size Comparison (Space complexity) [Log scale]

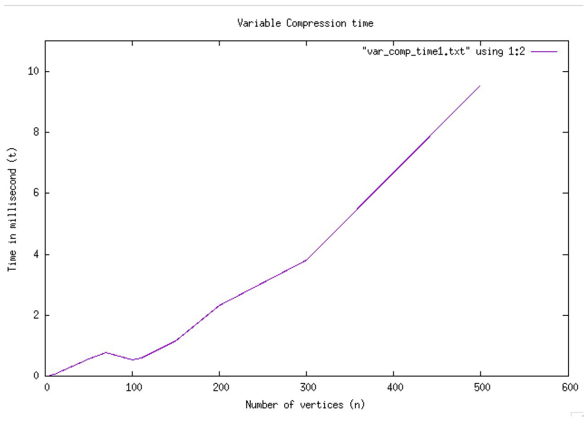


c) Size comparison (Space complexity) [Log scale]

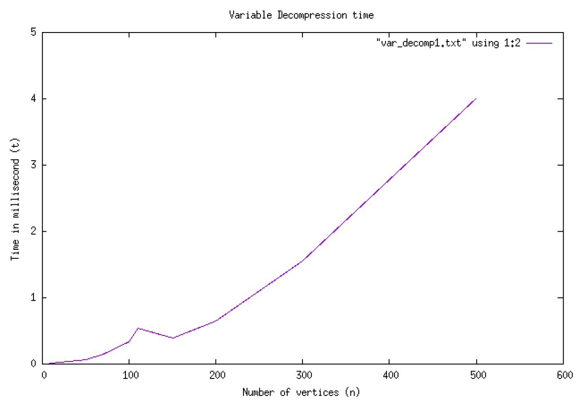


3) Bit-string method (Variable length)

a) Compression Time (Time complexity)



b) Decompression Time (Time complexity)



The Big-Oh complexities for all the methods are listed in the table below:-

Time Complexity of Conversion

Method	Best-case	Average-case	Worst-case
Base Conversion	$O(V^2)$	$O(V^2)$	$O(V^2)$
Bit-string (static)	$O(V^2)$	$O(V^2)$	$O(V^2)$
Bit-string (variable)	$O(V^2)$	$O(V^2)$	$O(V^2)$

Space Complexity

Method	Best-case	Average-case	Worst-case
Base Conversion	$O(V)$	$O(V)$	$O(V)$
Bit-string (static)	$O(1)$	$O(k)$	$O(V^2)$
Bit-string (variable)	$O(1)$	$O(k)$	$O(V^2)$

CONCLUSION

We observe that the ***Bit-string method*** performs well in all the cases. It doesn't have the problem of overflow as faced by the base conversion method. However, the implementation depends upon the architecture of the system (64-bit/32-bit). Thus we arrive at the conclusion that Bit-string method is the most superior one and most reliable, specifically the variable length implementation.

REFERENCES

[1] Crysttian A. Paixão, Flávio Codeço Coelho "*Matrix compression methods*" 1) Federal University of Santa Catarina, Curitibanos, SC, Brazil, 2)Applied Mathematics School, Getulio Vargas Foundation, Rio de Janeiro, RJ, Brazil.