Condensation of Adjacency matrix with parallel edges

Presented By:-
Maharshi Roy (ISM2014006)
Rajat Kumar Sahu (IHM2014501)
Amrit Daimary (IRM2014001)

Condensation of adjacency matrix

- Since the graph is undirected, we can perform the most trivial optimization of storing only upper or lower triangle of the whole matrix, cutting space occupied by half.
- We then propose 3 different algorithms to condense the graph without losing information:-
    A. Numerical Base Method
    B. Static Length Bit-string conversion
    C. Variable Length Bit-string conversion

Numerical base Method

1) In this method, we convert the matrix rows representing each vertex into a single integer by choosing an appropriate base.

2) The base is chosen as:-
   b=max(u1,u2,u3,,,un)+1
   I.e. 1 value greater than the maximum edge value.

3) Then each row is converted to a Single numerical value as:-
   val=u1+u2*b+u3*b^2+...un*b^(n-1)

4) This way we reduce our space from O(n^2) to O(n).

5) However, this method suffers From the obvious problem of Overflow.

```
//Choosing appropriate Base

Base = 0
for i = 1 to N
        for j=1 to i-1  //Iterate over all matrix elements
                Base = max (Base, graph[i][j])


//Condensation

for i = 1 to N
        val = 0
        for j = i-1 to 0
                val = val * Base + graph[i][j]

        condensed[i] = val       // A linear array


//Decondense

for i = 1 to N
        val = condensed[i]
        for j = 1 to i-1
                decondense[i][j] = val % Base
                val = val / Base
```

BIT-STRING METHOD (STATIC LENGTH)

1) In this method, we using 64 bit integers to store information.We try to utilize all 64 bit by not storing leading zeros for small numbers.

2) We find maximum bit length required to represent a number according given input.The maxlen is chosen as:-
   maxlen=max(u11,u12,u3,..,unn)
   uij=binary_length(graph[i][j])

3) Divide 64 bit in chunks of maxlen bit and store edge informations sequentially.

4) When unable to store upcoming edge information append next 64 bit in same fashion until all information store.

5) 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.

```
//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
        val ← val << offset //Right shift by
offset
        offset ← offset + maxlen
        x ← x | val  //logical OR
```

BIT-STRING METHOD (VARIABLE LENGTH)

1) In static length bit-string conversion, 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.

2) In this method we keep 2 separate set of integers :-
   length-string: stores the number of bits required by graph[i][j]
   value-string: stores value of graph[i][j]

3) We find
   maxbits = maximum number of bit required to store length of value of graph[i][j].

4) Divide 64 bit integer of length-string in chunks of maxbits and store length of value of edge informations sequentially.

5) Corresponding that length is allocated in value-string to store value of edge sequentially.

```
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
```

COMPLEXITY ANALYSIS

Time Complexity

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

Where k is the number of integers required