Unary Adder

This document tries to explain how the unary adder is incorporated into the program. The source code for k=1, 2, 3 can be found in `k1.cpp`, `k2.cpp`, and `k3.cpp` respectively.

Unary Adder for k=1

below is an example of a unary adder that counts the number of 1s in a given vector.

It uses nC2 auxillary variables where n is the length of the given vector

Pseudocode for unary adder

All array starts at index 1

```
//takes in an array 'a' which we want to count the number of ones of and returns the
unary counter
unaryCounter (array a):
    initialize 2 arrays: currentCounter and previousCounter
    currentCounter[1] <-> a[1]
    for i in range 2 to a.length :
        clear currentCounter
        currentCounter[1] <-> previousCounter[1] or a[i]

        for j in range 2 to i-1:
            currentCounter[j] <-> previousCounter[j] or (previousCounter[j-1] and
a[i])

        currentCounter[i] <-> prevoiusCounter[i-1] and a[i]
        previousCounter = currentCounter //sets up for next iteration

    return currentCounter
```

Example and C++ code

Example of rough idea:

given `[0 1 0 1 1]`, the adder looks like

```
0
1 0
1 0 0
1 1 0 0
1 1 1 0 0 //this is the final result that signals there are 3 1s in the vector
```

```
/*
@brief : counts the number of 1s in a vector with a unary counter
@param : a : a vector that is a row of the adj matrix
         start: the starting index to allocate auxilary variables
@return : returns the index of the next available variable
*/
vector<int> k1(vector<int> a, int &start){
```

```
    //pre: partial sum up to the previous index (used to calculate current partial
sum)
    //cur: current partial sum
    //both have the form 111..100..0 where the # of 1s represents correspond to the
number of ones up to the current index
    //can also be seen as a sorted row
    vector<int> pre, cur;

    //current var corresponding to whether the first index is a 1
    iff(start, a[0]);

    //push the partial sum up to first index into pre (with offset to avoid using
index 0 of pre)
    pre.push_back(-1);
    pre.push_back(start++);

    //compute the partial sum up to index i of a
    for(int i = 1; i< a.size(); i++){
        cur.clear();
        cur.push_back(-1);

        //current partial sum is 0 if previous sum is 0 and a[i] is 0.
        //cur[1] <-> pre[1] or a[i]
        print2(-pre[1], start);
        print2(-a[i], start);
        print3(-start, pre[1], a[i]);
        cur.push_back(start++);

        //loop all the possible 1 count: 1 to i
        for(int j=2;j<=i;j++){
            //the jth bit of current partial sum is 1 if it was 1 previously or it
was 1 in the j-1th bit and a[i] is 1
            //cur[j] <-> pre[j] or (pre[j-1] and a[i])
            int b = a[i], c = pre[j], d = pre[j-1];
            print2(-c, start);
            print3(-b, -d, start);
            print3(b, c, -start);
            print3(d, c, -start);
            cur.push_back(start++);
        }
        //cur[i+1] <-> pre[i] and a[i]
        print2(-start, pre[i]);
        print2(-start, a[i]);
        print3(start, -pre[i], -a[i]);
        cur.push_back(start++);

        //set pre to cur for the iteration
        pre = cur;
        assert(pre.size()==i+2);
    }
    pre.erase(pre.begin());
    return pre;
```

```
}
```

Ensuring that the top row is the best

Given two unary counters x and y, instead of using lex, simply ensuring that there is no index such that x[i] = 1 and y[i] = 0 ensures that x <= y since we already know that they are sorted with all 1s in the front.

This is achieved with the `cardLeq` function:

```
void cardLeq(vi a, vi b){
    for(int i=0;i<a.size();i++){
        print2(-a[i], b[i]);
    }
}
```

How the adder is used to ensure k=1

1. compute the counter for row 1
2. for each subsequent row:
    1. compute the counter for that row
    2. use `cardLeq` to ensure that counter for first row is less than or equal to counter for second row.

```
    vi topk1 = k1(row[1], idx);

    for(int i=2;i<=n;i++){
        //curk1 is the unary adder that stores the k=1 constraint of the current
row.
        vi curk1 = k1(row[i], idx);
        // lex(topk1, curk1, idx);
        cardLeq(topk1, curk1);
    }
```

K=2 with unary adder

The idea is the same with a few tweaks.

Given arrays a and b, which are two rows of the adjacency matrix:

1. declare auxillary variables p1[i] <-> a[i] = 0 and b[i] = 1
2. count p1 with the unary counter, name the returned counter `counter01`
3. declare auxillary variables p2[i] <-> a[i] = 1 and b[i] = 1
4. count p2 with the unary counter, name the returned counter `counter11`
5. concatenate `counter01` and `counter11` then return the concatenated array

```
//returns a vector of (0, 1) counter concatenated with (1, 1) counter
vi k2(vi a, vi b, int &st){
    vi ans;
    vi cnt01, cnt11, counter01, counter11;
```

```
    //writes variables to indicate (0, 1) pairs
    FOR(i, 0, a.size()-1){
        //st <-> -a[i] and b[i]
        //cnf: (A ∨ ¬B ∨ St) ∧ (¬St ∨ ¬A) ∧ (¬St ∨ B)
        print3(a[i], -b[i], st); print2(-st, -a[i]);  print2(-st, b[i]);
        cnt01.pb(st++);
    }


    //writes variables to indicate (1, 1) pairs
    FOR(i, 0, a.size()-1){
        //st <-> a[i] and b[i]
        //cnf: (-A ∨ ¬B ∨ St) ∧ (¬St ∨ A) ∧ (¬St ∨ B)
        print3(-a[i], -b[i], st); print2(-st, a[i]);  print2(-st, b[i]);
        cnt11.pb(st++);
    }
    //pass (0, 1) vector into binary adder
    counter01 = k1(cnt01, st);
    assert(counter01.size()==n);
    //pass (1, 1) vector into binary adder
    counter11 = k1(cnt11, st);
    assert(counter11.size()==n);

    ans.insert(ans.end(), counter01.begin(), counter01.end());
    ans.insert(ans.end(), counter11.begin(), counter11.end());
    return ans;
}
```

Ensuring k=2

The only subtle part is the introduction of variables p1 and p2 which makes sure we check k=2 only when necessary, which is:

1. the two nodes are not adjacent, and
2. the first row of the current pair has the same number of ones as the first row

```
    /*
        encode k=1 for first row
    */
    vi topk1 = k1(row[1], idx);

    /*
        encode k=2 additional constraints for first two rows
    */
    vi topk2 = k2(row[1], row[2], idx);


    for(int i=1;i<=n;i++){
        /*
            encode k=1 for row i of adj matrix
        */
```

```
        //curk1 is the unary adder that stores the k=1 constraint of the current
row.
        vi curk1 = k1(row[i], idx);
        cardLeq(topk1, curk1);

        //p1 <-> row[1] and row[i] have same ardinality
        int p1 = checkEqual(topk1, curk1, idx);

        for(int j=1; j<=n;j++){
            if(i==j) continue;
            if(i==1&&j==2) continue;

            /*
                encode k=2 for row i , j
            */
            vi curk2 = k2(row[i], row[j],  idx);

           int p2 = idx++;
           //p2 <-> (i, j are adjacent) or (p1 is false), in which case we don't
need to check anymore
           //(p1 ∨ P2) ∧ (¬Flat ∨ P2) ∧ (¬P2 ∨ -p1 ∨ Flat)
           print2(p1, p2); print2(-flat(i, j), p2); print3(-p2, -p1, flat(i, j));

            /*
                compares the k=2 encoding of first two rows and current pair of rows
iff p2 is false.
            */
           cardLeq(topk2, curk2, p2); //an overloaded version of cardLeq that only
reinforces if p2 is false
        }
    }
```