More Hints on Implementing the Dynamic Programming Solution to the Traveling Salesperson Problem.

First, I repeat the suggestions on Page 5-7 of the SetBitVectors pdf document already posted.

In this last paragraph, it looks like a bottom-up computation starting with small subsets and working up using larger sets and using results from previous iterations. This is similar to computing factorial of n iteratively. Consider the following C++ iteratively solution for factorial of n.

```
unsigned long factorial(unsigned n) {
        unsigned long result = 1;
        for( unsigned int k = 2; k <= n; k++ ){
                        result = k*result;
        }
        return result;
}
```

This is similar to the textbook TSP-dynamic-programming algorithm show above in the following way. With factorial implementation below we start with a small number (=1) and then construct bigger values in a bottom-up computation. With the TSP algorithm given in the textbook, it is potentially challenging to construct all the subsets, A, each of the same size k for the kth iteration. It can be done as described earlier. Perhaps, there is an alternative way to do this bottom up. Consider the alternative recursive solution for solving the factorial of n.

```
unsigned long factorial(unsigned n) { return (n <= 1)? 1 : n * factorial(n-1); }
```

One could recursively compute the sub-tour costs in a similar way. Start with n-1 vertices and work downward until we could get to simple single edges as is computed in the first loop:

```
for ( i = 2; i <= n; i ++)
  D[i][Ø] = W[i][1];
```

We can still do this. But, we go downward from the initial case and get to the case of looking up D[i][Ø] for each i when needed. This is similar to the divide-and-conquer approach. Here we construct the costs on the way up and use other already min-computed sub-paths that used the same set of vertices to vertex 1.

We could alter the textbook algorithm as follows. The parameter name S (of Set type) will be used so as not to be confused with set A in the textbook algorithm.

```
/*compute the best cost from i to 1 via vertices in S*/

unsigned int computeMinTourCost(vertex i, set S) {
        unsigned int aCost;  /* emphasize that this is a local variable */
        vertex best_vj;
        boolean found_at_least_one_vertex = false;

        if S = ∅ the return D[i][∅]
        if D[i][∅] >= 0 then return D[i][S]
        bestCost = Max_Int;  /* could remain assigned to this if no edge connected to the other vertices */
        best_j = -1
        for (each j in S ){   /* this if is essentially iterating over the set S */
                aCost = D[i][j] + computeMinTourCost( j, S – {j} );
                if ( aCost < bestCost ) {
                   found_at_least_one_vertex = true;
                    bestCost = aCost;
                    best_j = j;
                }
        if( found_at_least_one_vertex ) {
                D[best_j][S] = bestCost;
                P[i][S] = best_j;
        }
        return bestCost;
}
```

The top-level call could look something like the following; it may be different for your implementation.

unsigned int bestTourCost = *computeMinTourCost*(1, V – {1});

Of course if you are using bitset<N> in C++, you might write something like this:

```
unsigned int bestTourCost = computeMinTourCost( 0, V – {0});
```

Remember to make all the adjustments to indices in your definition of *computMinTourCost*(vertex,set);

Note: The data type set can be replaced with `bitset<N>` with N set to 32. Also, it is efficient to pass the bitset<32> parameter value. Also remember that the D, W, and P matrices have to be accessible by all activations of computeMinTourCost.

Note that in this function *computeMinTourCost* is recursive. This helps with the iteration over each set A and setting each v_j. In the sample code below, in order to distinguish it from the book's algorithm, the C++ code names have been changed. The top-level call to the book's *travel* function is changed to gCost and gCost also represents the *computeMinTourCost* in the above pseudo-code.

The gtable below is corresponds to the D matrix of V x PowerSet(V), where V are the vertices with the indices 0, 1, … n-1. Note the shift from 1..n. This code below also shows how to allocate the two-dimensional matrix.

C++ Partial Code. This is intended to show you how to build the D table which is called `gTable` in the code below.

```
class tspProblem{
private:

         vector<vector<int>>  gTable;  //allocate memoization table and initialize to -1
         vector<vector<unsigned int>> pathTable;
         bitset<32> S;

         unsigned int gCost(unsigned int i, bitset<32> S, Graph &g)
         {
                  unsigned int nVertices = g.getNumVertices();
                  unsigned int costThru_j;
                  unsigned int min_j;
                  bitset<32> min_S;
                  if (S.none()) return g.getEdgeCost(i, 0);
                  if (gTable[i][S.to_ulong()] >= 0)return gTable[i][ S.to_ulong()];  //if >=0, min distance already computed

                  //Reached here so that we can compute g(i,S) and iterate over all vertices in this instance of S.
                  unsigned int answer = INT_MAX;
                  bool min_j_found = false;
                  for (unsigned j = 1 ; j <  nVertices; j++){  //j = 1 since vertex 0 is not in S, j < nVertices
                                                     // since one of the each non-start/final vertex is removed from S
                           if (S.test(j)) { //if S.test(j) is 1 then j is in S.
                                   bitset<32> S_Temp = S; //get a temporary set with S-{j}. That is, j is removed from S
                  // YOU CONTINUE TO ADD CODE HERE

         }
public:
         tuple<int, vector<string>> solve(Graph & g)
         {
                  vector<int> v_temp;    //a constructed row for gTable[i][S] for a given vertex i using all subsets of graphs, vertices
                  vector<unsigned> v_pathTemp;  //a constructed row for pathTable for a give vertex i using S to start/final vertex
                  unsigned int nVertices = g.getNumVertices();
                  unsigned int minCost = 0;
                  for (unsigned int i = 0; i < nVertices; i++){
                           v_temp.resize(static_cast<int>(pow(2, nVertices)));  //allocate memoization table and initialize to -1
                           v_temp.assign(v_temp.size(), -1);   // use -1 for unused entries
                           gTable.push_back(v_temp);           //create a next row in gTable.
                  }
                  //build path table also
                  //Start S as the full set of vertices of graph g. So, all bits are set to 1. vertex i has bitposition i.
                  for (size_t i = 0; i < nVertices; i++){
                           S.set(i);
                  }

                  minCost = gCost(0, S.reset(0), g);   //Use vertex 0 as starting/ending vertex.
                                                       //S.reset(0) removes vertex 0 from S. S-{0}.

                  //start solving the problem
                 // YOU CONTINUE TO ADD CODE HERE


         }//end solve member function

}; //end tsp class
```

```
// You also need a main program driver that prompts the user for the file name opens the
file and then makes the calls to graph member functions and the tsp object solve member
function.
```