Set Review and Bitvector (bitset) Representation as Used Algorithms using Dynamic Programming for Traveling Salesperson Problem

Cardinality of a Set: The number of elements in a set, also called the *cardinality* of a set S. Notation: |S| = n.

Examples: |{1,2,3,4}| = 4; |{1,3,4}| = 3; |{a,d}| = 2.

Subset: A ⊆ B if and only if for all x ∈ A, then x ∈ B.

Set Equality: A = B iff A ⊆ B and B ⊆ A.

Example: {1,2,3} = {1,3,2} = {2,1,3} = {2,3,1} = {3,1,2} = {3,2,1}.

Sets have no default ordering.

Powerset: Powerset(S) = {A | A ⊆ S}. That is, Powerset(S) is the set of all subsets of S. Often, S is called the Universal set.

Example:
Powerset({1,2,3,4}) = { ∅, {1}, [2}, {3}, {4}, {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4}, {1,2,3}, {1,3,4}, {2,3,4}, {1,2,3,4} }

Observation: For a set S, where |S| = n, the number of subsets of size k (0≤k≤n) = $\binom{n}{k}$.

Example using set {1,2,3,4}: $\binom{4}{2} = 6 =$ |{1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4}}|. There is one empty set: $\binom{4}{0} = 1$. There is one full set {1,2,3,4}: $\binom{4}{4} = 1$. We add up $\binom{4}{0} + \binom{4}{1} + \binom{4}{2} + \binom{4}{3} + \binom{4}{4} = 1 + 4 + 6 + 4 + 1 = 16$.

In general, we add up all the cardinalitys of all possible subsets of a set S, we have: $2^n = \sum_{k=0}^{n} \binom{n}{k}$.

For each set in Powerset(S), there is a corresponding index value: 0, 1, 2, …, 2^n-1. One way of representing sets that are limited to a Powerset(S), where |Powerset(S)| = 2^n where n is a reasonable size, we can use a bitvector of size n and then interpreting the bitvector as a base-2 representation of a number, where the rightmost binary digit is the “ones” place ($2^0 = 1$)and the leftmost binary digit is the most significant digit and has weight 2^{n-1}. Recall, $\sum_{i=0}^{n-1} 2^i = 2^n - 1$.

With a set S = {1,2, …, n}, the corresponding bitvector for set S is a vector with index range 1..n and component type having values 0 or 1. Usually, we draw a bitvector object from right to left. (Reason will be apparent later.) For a subset A ⊆ S, and corresponding bitvector bvA[1..n],
bvA[i]= 1 iff i ∈A and bvA[i] = 0 iff i ∉ A.

bvA[1..n]:

n	…	2	1
0 or 1	…	0 or 1	0 or 1

Suppose we have S = {1,2,3,4}. For a set A = {1,3,4} the bitvector instance is:

4	3	2	1
1	1	0	1

Each bitvector instance can be interpreted as a base-2 representation of an unsigned integer

Example: The bitvector representation of {1,3,4} is shown below. It can can be interpreted as the number thirteen or (13 in base-10).

4	3	2	1
1	1	0	1

Example: Suppose S = {1,2,3,4}. All sets in Powerset(S) along with their bitvector and number representation:

Set	bitvector (bitset)	Base-2 interpretation
	<4321>	
∅	0000	0
{1}	0001	1
{2}	0010	2
{2,1} = {1,2}	0011	3
{3}	0100	4
{3,1} = {1, 3}	0101	5
{3,2} = {2, 3}	0110	6
{3,2,1] = {1,2,3}	0111	7
[4}	1000	8
{4,1} = {1,4}	1001	9
{4,2} ={2,4}	1010	10
{4,2,1} = {1,2,4}	1011	11
{4,3} = {3,4}	1100	12
{4,3,1} = {1,3,4}	1101	13
{4,3,2} = {2,3,4}	1110	14
{4,3,2,1}	1111	15

In C++, the Standard Template Library (STL) `bitset` implements the bitvector structure. It is a template class with a template parameter of `size_t`. The type, `size_t,` is synonymous `unsigned int`. The template's actual parameter would be the value 4 if we are to implement the above example of {1,2,3,4}. An example of a set object declaration/definition would be `bitset<4> s`. The `bitset` type in STL uses indices from {0,…,n-1} instead of {1,…,n}. The `bitset<N>` class orders the bit indices from right to left, where bit index 0 is considered the rightmost bit and n-1 considered the leftmost bit. The [i] operator accesses the value at the i position where $0 \leq i \leq n-1$. The member function `test(i)` will also do the same and will generate an `out_of_range` exception; [] will not. In either case, finding out if an element is in a bitset take $\Theta(1)$ time, constant time. For each bitset object value, which is a set, one can get the corresponding `unsigned int` associated with the set when interpreted as a base-2 number. These are: `to_ulong` or `to_ullong`. Adding an element **i** from the universal set to a set A, just bind A[i] to 1. If A[i] it is already set to 1, then it remains 1, which is the same semantics of when adding an element to a set that already has that element. Similarly, removing an element from A, just bind A[i] to 0. For the full list of operations, please see, http://www.cplusplus.com/reference/bitset/bitset/ . There are other sites, e.g. MSDN. If you need *set* objects as an indices to a general vector or matrix, you can use vector<bitset<N>> has index range of 0..2^N-1. Since vectors and matrices use 0..m-1 index ranges, one can use `to_ulong` member functions to get a value between 0..m-1, where m = 2^N.

If one is representing a universal set with S = {1,2,… n}, then one will have to adjust the element values for the STL `bitset` object range by subtracting 1 for each element (which becomes an index for a bitset object). If you are using a set of strings, such as a set of city names, you will have to assign an index internally in your program to each city name by storing the names in a `vector<string>`. The index of a vector object can be used. When listing elements of a set for output, one would use the indices in the set that are assigned to 1 to find the corresponding index for the `vector<string>` object representing the names of cities.

Some notations, assumptions, and observations for implementing bitvectors and the STL bitset<N> representation:

- Assume Universal Set = U = {0, 1, 2, …, n-1}
- Assume S ⊆ U, or, equivalently, S ∈Powerset(U).
- Let BV(S) be the bitvector representation of set S.
- If |U| = n, the size(BV(U)) = n, and BV(U) = <1,1,…, 1>. For all S ⊆ U, size(BV(S)) = n.
- Let nBV(S) = the base-2 interpretation of the bitvector of S.

Observations:

- For i ∈ U, nBV({i}) =2^i.
- For ∅, nBV(∅) = 0.
- For U, where |U| = n, nBV(U) = $2^n - 1$

Let S ⊆ U , T ⊆ U, |U| = n.

operations on sets S, T	return type	abstract implementation	&: bitwise and | : bitwise-or ~:bitwise complement $(2^n - 1)$: bitvector of n 1's. 2^i: bitvector of n-1 0s, i-th bit = 1	STL bit indices <n-1,.. 0> 0 is rightmost n-1 is leftmost `bitset<n> s,t`
membership i∈ S	boolean	S ∩ {i} == {i}	(nBV(S) & nBV({i}) == nBV({i}	`s.test(i)` `s[i]`
non-member i ∉ S	boolean	S ∩ {i} == ∅	(nBV(S) & nBV({i}) == 0	
S ⊆ T	boolean	S ∩ T == S	(nBV(S) & nBV(T)) == nBV(S)	`(s & t) == s`
S ∩ T	set	S ∩ T	nBV(S) & nBV(T)	`s & t`
S ∪ T	set	S ∪ T	nBV(S) | nBV(T)	`s | t`
S - T	set	S - T	nBV(S) | ~nBV(T)	`s & (~t)`
~S	set	U - S	$(2^n - 1)$ & ~nBV(S)	`~s`
add i to S	set	S U {i}	nBV(S) | nBV({i})	`s.set(i)`
remove i from S	set	S – {i}	nBV(S) & ~(2^i)	`s.reset(i)`
S == U	boolean	S ∩ U == U	(nBV(S) & nBV(U)) == nBV(U)	`s.all()`
S == ∅	boolean	S == ∅	nBV(S) == 0	`s.none()`
S != ∅	boolean	S != ∅	nBV(S) != 0	`s.any()`
|S|	{0,…, n-1}	|S|	number of bits set to 1	`s.count()`
|U|	{0,…, n-1}	|U|	number of bit positions for bitvector	`for any s` `s.size()`

Dynamic Programming Based Algorithm for Traveling Salesperson Problem. Taken from *Foundations of Algorithms* Textbook, 5th edition, by Richard Neopolitan.

```
void travel(int n, const number W[][], index P[][], number& minlength)
{
        index i, j, k;
        number D[1..n][subset of V − {v1}];
        for ( i = 2; i <= n; i ++)
          D[i][∅] = W[i][1];

        for( k = 1; k <= n – 2; k++)
           for( all subsets A ⊆ V−{v1}containing k vertices)
             for ( i such the i ≠ 1 and vi not in A ){
                D[i][A] = minimum_{2≤j≤n}(W[i][j] + D[j][A−{vj}] );
                P[i][A] = the value of j that gave the minimum.
             }
        D[1][V − {v1}] = minimum_{2≤j≤n}(W[1][j] + D[j][V−{v1, vj}] );
        P[1][ V − {v1}] = value of j that gave the minimum;
        minlength = D[1][V − {v1}];
}
```

For the two key data structures, matrices D and P, we need subsets of a universal set $V - \{v_1\}$ as "indices" for the second dimension. The first[] will select the row of these matrices and the second [] will select the column of these matrices. Vectors and matrices require index values 1..n, or, in the case of C++, 0..(n-1). To use sets as indices we need a one-to-one correspondence (or bijection) between every subset of $V - \{v_1\}$ and 1..n. For a C++ implementation, we can use the `bitset<N>` class for the implementation and use the `to_ulong()` member function in `bitset<N>` class to get the corresponding number associated with each set. To get the set from the number, we can use the `to_string()`member function, but that will not be used here. For a C++ implementation, the above algorithm will need to be modified by changing the indices from 1..n to 0..(n-1). For now, let's use the index range of 1..n for the purpose of understanding the algorithm as presented above.

The first for-loop of setting D[i][∅] = W[i][1] is saying every vertex (except for v_1) the cost of edge $(v_i\ v_1)$ must be considered. Remember the cost each edge (i,j) are stored in W[i][j]. Each edge cost for $(v_i\ v_1)$ will be eventually checked to see if the edge will be the last edge in the minimum tour. The ∅ means go directly from vertex i to vertex 1 without going through any other vertex. Think of this as the bottom of the bottom-up computations and we want to "remember" this and store this value in the D[][] table.

We have solved the problem of representing subsets for the column indices for D and P. The next implementation problem is generating the subsets, A, in the second loop of the three nested loops. Here are the three loops:

```
for( k = 1; k <= n – 2; k++)
   for( all subsets A ⊆ V−{v1} containing k vertices)
     for ( i such the i ≠ 1 and vi not in A ){
        D[i][A] = minimum_{2≤j≤n}(W[i][j] + D[j][A−{vj}] );
        P[i][A] = the value of j that gave the minimum.
     }
```

Suppose V = {1,2, …, n}. For each value of k from the outermost loop, we have to generate a subset of V−$\{v_1\}$ = {2, 3, …, n} of k vertices. We continue to do this with each k, but only up to $(n-2)$ rather than $(n-1)$. Why? See the final two statements in the algorithm following the three-nested for-loops.

With k = 1, we need all the subsets of V−$\{v_1\}$ = {2, 3, …, n} that have only 1 vertex. So, we need $(n-1)$ singleton sets: {2}, {3}, …, {n}. When k=2, we need $\binom{n-1}{2}$ sets: {2, 3}, {2, 4}, …, {2, n}, {3, 4}, …, {3,n}, …, {n-1,n}. In general, we need $\binom{n-1}{k}$ subsets of size k, $1 \le k \le (n-2)$. For the last iteration for k, we have k=$(n-2)$ and we have $\binom{n-1}{n-2}$ subsets each of size $(n-2)$ chosen from the set {2,3,…, n}. Each set has one of the numbers missing: {3, …, n-1}, {2, 4, … n-1}, …, {2, 3, …, i-th number missing, …, n-1}, … , {2, 3, …, n-2}. This last iteration generates $(n-1)$ sets. Overall, the second for-loop has to generate: $\binom{n-1}{1} + \binom{n-1}{2} + \cdots \binom{n-1}{n-2}$ subsets. Recall, $2^n = \sum_{k=0}^{n} \binom{n}{k}$. So, $\sum_{k=1}^{n-2} \binom{n-1}{k} = 2^{n-2} - 2.$ To fully analyze the three nested loops, the basic operation of the body of the most nested loop must consider n-1-k when considering each set A containing k vertices an this must be done k times. See the textbook for solving T(n) = $\sum_{k=1}^{n-2} (n-1-k)(k)\binom{n}{k} = (n-1)(n-2)2^{n-3}$. This is better than Ω(n!).

So, we know how many subsets, A, have to be generated in the second loop of the nested loops as a function of the current value of k and how many times the basic operation of the innermost body is executed. The problem is generating all $\binom{n-1}{k}$ sets in the second loop and then iterating over them in the third, most nested loop. One could go through all the subsets of V−$\{v_1\}$ and pick out the ones with cardinality of k = 1. The next iteration for k (=2), go iterate through all of the subsets and pick out the ones with two elements, …, continue, and stop when k = n-1. For each k, we have to store all of the constructed sets (the A's) in a temporary list or vector and then execute the innermost loop.

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 = Ø the return D[i][Ø]
        if D[i][Ø] >= 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.

Some other implementation suggestions:

- Whether you implement the textbook's algorithm or the one presented in these lecture notes is up to you.
- In order to access the D, W, and P matrices without making them global, you must put them in a class and set the access rights to `private`.
- You might consider making a tspProblem class. Each class instance really becomes a computation problem instance. You could make a solve member function. The input (parameter) would be a graph object. However, this means you need a graph class.
- In your textbook, the input is a graph and n for the number of vertices. With a graph class with constructors and/or read() member functions and the number of vertices would be assigned implicitly.
- Having a graph class with a read() and print() function would be member functions. The read() member function could read an input file and construct the W matrix, a private data member of the graph class. One would write a member function with getedgeCost(i,j) that would return the cost of edge (i,j) and would return the value of W[i][j].
- The read() and print() parameters are, respectively, `ifstream&` and `ofstream&.` The default parameters could be `cin` and `cout`, respectively.

These lecture notes are intended to guide you in your solution and implementation of algorithm using the dynamic programming approach.