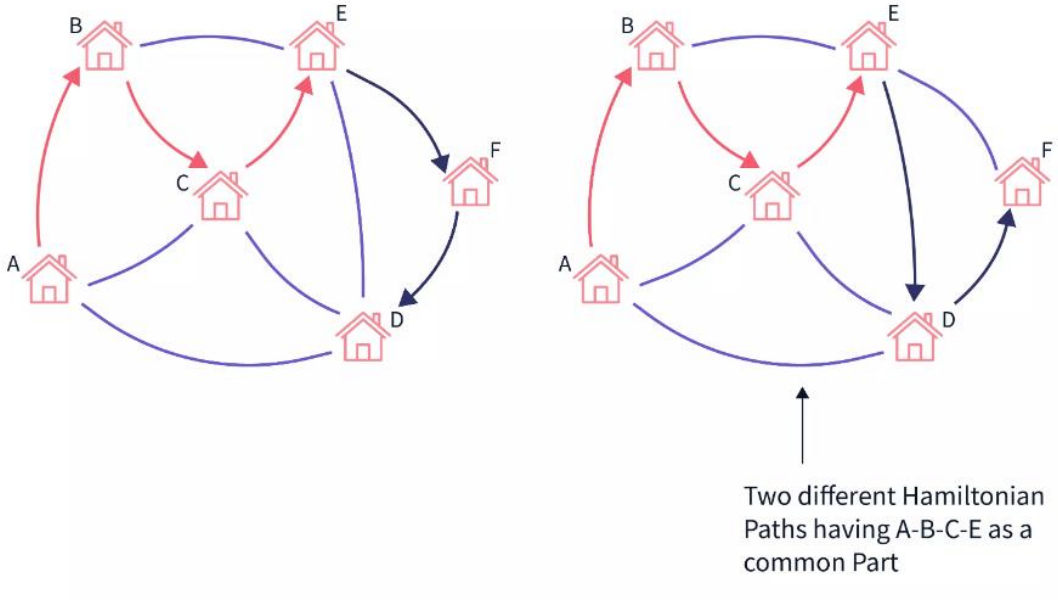
3) Dynamic Programming Approach

In this approach as the name suggests we need to store the memos (memo can be interpreted as memory) of the small solutions to build larger solutions, but the question is what are those small problems and their solutions to store and how to build a memo based solution for the **Hamiltonian Path**, we will answer all these questions in the following section

Consider again the same problem of **Ray** visiting the houses of his friends in his neighborhood, in the naive approach we simply had to generate all paths and check for each of them whether it is a **Hamiltonian Path** or not but in this solution we are doing a lot of redundant work like for example refer to the image below:

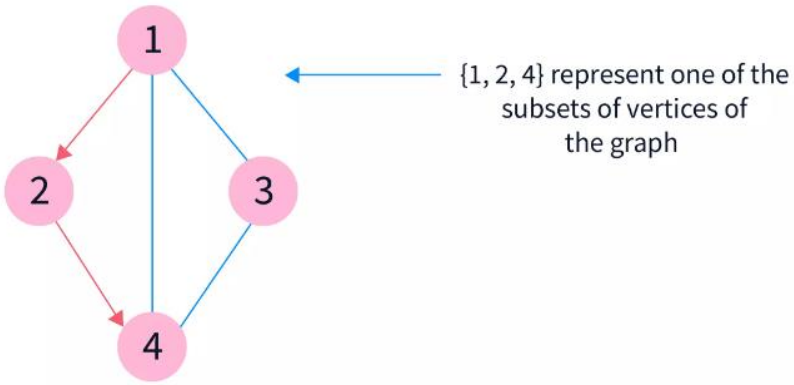


Two different Hamiltonian Paths having A-B-C-E as a common Part

In the figure above, the two **Hamiltonian Paths** **{A, B, C, E, F, D}** and **{A, B, C, E, D, F}** are separately generated by the previous approach, but we can observe that the part **{A, B, C, E}** (Marked with red edges) is common to both Paths which is redundant work. This can be stored in a memo and quickly used while building the paths that visit the next unvisited nodes like **{A, B, C, E, D}** and **{A, B, C, E, F}**. In these paths, we simply added edges **E-D** and **E-F** to the previously memoized path **{A, B, C, E}**, which can be quickly retained because we stored it previously, and these new paths can also be memoized in the same way we did for **{A, B, C, E}**. And so on, until we visit all the vertices of the graph exactly once, and once this happens, we are already done with the problem.

Now, using the above idea of storing partial paths and building the solution towards final **Hamiltonian Paths**, we shall be emphasizing the questions like implementing the above strategy in practice in the best possible way and the working solution of the problem, so let's jump to the implementation part.

Implementation of Dynamic Programming Approach: We have learned about partial solutions or partial paths in this specific case, we can conclude that those partial paths are simply the subsets of the vertices of the graph where there is a path in the subset that visits every single vertex in it and ends at some vertex, in other words, we build larger subsets(Hamiltonian paths) from the smaller subsets(Hamiltonian paths)
Let's understand the idea with the following example:



In this graph, **U = {1,2,4}** is one of the **16** subsets of the vertices of the graph. In this subset we observe that there exists a path that visits all vertices and ends at **4**. Now, let's consider another subset i.e. **V = {1,2}** and ask ourselves does there exist a path that visits every vertex in **V** exactly once and ends at **2**, the answer is yes there exists such path. Now let's get back to the previous subset **U** which is just **V + 4** (here **+** represents the addition of vertex in the subset).

We know at this point that **V** is a subset where there is a path that visits each and every vertex exactly once and it ends at **2**, simply extending this vertex **2** to add **4** to make the subset **U**. Now, **U** is a subset which contains a path that visits every vertex exactly once and ends at **4**.

Hence, we built the larger subset **U** with a valid path from a smaller subset **V** with a valid path and in the implementation, we start processing from smaller subsets and build larger ones. The next question is that how to represent those subsets, one of the ways is to use bitmasking. Let's understand the Pseudocode implementation:

```
function hamiltonian_Path(graph[][],n)
    for i = 0 to 2^n
        for j = 0 to n
            dp[j][i] = false
    for j = 0 to n
        dp[j][2^j] = true

    // Iterating over all possible subsets of vertices
    for i = 0 to 2^n
        for j = 0 to n
            // if jth vertex is included in current subset
            if jth bit is set in i
                // find a neighbor k of j, also present in the current subset,
                // such that the subset `i XOR 2^j` can be extended to i
                for k = 0 to n
                    if j != k and kth bit is set in i and graph[k][j] == true
                        if dp[k][ i  XOR 2^j ] == true
                            dp[j][i]=true
                            break
    for i = 0 to n
        if dp[i][(2^n)-1] == true
            return true
    return false
```

In the above implementation, **graph**[][],**n** represents the given graph represented in the adjacency matrix form and the number of vertices in the graph. $dp[n][2^n]$ is a boolean matrix is used to check for the existence of **hamiltonian paths** in different subsets which are represented by bitmasks while there are total 2^n subsets including the empty subset(because in bitmask of size **n**, there are two ways either to select a vertex or not), **dp**[**j**][**i**] represents the existence of the path in mask(subset) **i** that ends at **j**.

> **Note:** In a mask **i** of size **n**, the **kth** setbit represents that **kth** vertex is present in the subset represented by this mask.

The first loop iterates from **i = 0** to $i = 2^n - 1$ i.e. from subset containing no vertex to the subset containing all vertices and for each subset **i** the inner loop iterates all vertices and initialize the **dp**[**j**][**i**] to false because no subset is processed yet, this is called initialization.

The next loop iterates from **j=0** to **j=n-1** i.e. all vertices(notice we are considering 0-based numbering here) and the subset/mask 2^j will simply represent the subset containing only the vertex **j**, as **j** is set in it and obviously, it will be true i.e. a subset containing a single vertex is always valid hence we make them true.

The next loop iterates over all subsets from **i = 0** to $i = 2^n - 1$ and for each subset represented by the mask **i** there is another inner loop which iterates over all vertices of this subset from **j=0** to **j=n-1** and it checks if **jth** bit is set in **i** means if **jth** vertex is included in **i**, then there is another inner loop from **k=0** to **k=n-1** which checks for the neighbours of **j** which are present in the subset **i**, i.e. there is a direct edge between **k** and **j**, then we check for the subset which contains all vertices which are in **i** except **j** i.e. **i - {j}** and there is a path that visits every vertex in **i - {j}** and ends at **k**, it is checked by $dp[k][iXOR2^j]$. Here, **XOR** operator simply removes the **jth** vertex from the subset **i** and we are left with the subset **i - {j}** which contains a path visiting every single vertex exactly once and ending at **k**.

Now, if $dp[k][iXOR2^j]$ is true, then we extend the edge from vertex **k** to **j** and hence, make **dp**[**j**][**i**] to be true. Thus, we keep building the larger solutions and marking the **dp** table for the existence of the **Hamiltonian path**.

Finally, We iterate from **i=0** to **i=n-1** and check if $dp[i][2^n - 1]$ is true, if it is then we return true because $2^n - 1$ is the subset containing all vertices and hence represents a valid **Hamiltonian path** is $dp[i][2^n - 1]$ is true.If we do not find any such path in the graph we return false.

Now, let's analyze the time complexity of this approach:

Time Complexity: It can be seen that we are iterating over all subsets that take $O(2^n)$ and for each subset, we iterate over all vertices of this subset and then we use an inner loop to check for the edges and this takes $O(n^2)$ and the overall complexity is $O(2^n * n^2)$.