For an example, for a string src\_str, we want to transform it to dest\_str, and the dictionary is DICT.

The basic idea is taking recursive actions to get the optimal path. That is, we assume the first transformation occurs on the ith character of src\_str and the character is transformed from src\_str[i] to new\_c (thus the new src\_str is src\_str' now). If src\_str' is in the dictionary, we then recursively call the routine to get the optimal path from src\_str' to dest\_str. We try all positions of src\_str and all transformed characters from ' a' to ' z', then we can get the optimal path from src\_str to dest\_str.

During the recursive call, we should put the transformed string into a hash table, so that later we should not visit it again to avoid dead loop. One problem with this method is that it may have quite deep call stack and extremely computation complexity.

To illustrate this, we can treat the problem as a tree traverse procedure, as described in following figure 1. The src\_str is the root (i.e., A). First we transform the 0th character of A to a new one (and the new str is B now) and it is in the dictionary. Then we apply the same procedure to B, which makes us goes to next level D …… When returning back to A, we now transform the 0th character of A to another new one (and the new str is C now) and it is in the dictionary. Then we apply the same procedure to C, which makes us goes to next level F ……. We can see we will traverse each subtree to the very deep end end then can we get the final optimal result. Besides, some sub trees may be the same so we do repeatedly computation. Now a clever boy would say: ”Whenever we get the optimal path of the tree with root R, we can record its optimal path and later when we visit R again, just taking use of the answer in the record”. This seems right but actually it is wrong because the subtree with the same root string may have different structures.

Let us see Figure 2. We want to get the optimal path from src (A) to dest. We first transform A to E. Then we continue the transformation from deep depth to high depth. When we transforms to B, we cannot continue transform B to C because C is already on the path. So we transform B to I and continue. We can get the optimal path from B to dest is B->I->J->K->Dest and record this result. After the recursion returns to A, A continues to try to transform to B. If we count on the previous record, we can directly get the optimal path from B to dest is B->I->J->K->J. But the actual optimal path is B->C->Dest. If we indeed want to use this look up method, the key in the record should not be B itself, it should be the path AECGB. To be specific, LookupTable[AECGB] = BIJKDest and LookupTable[AB] = BCDest.

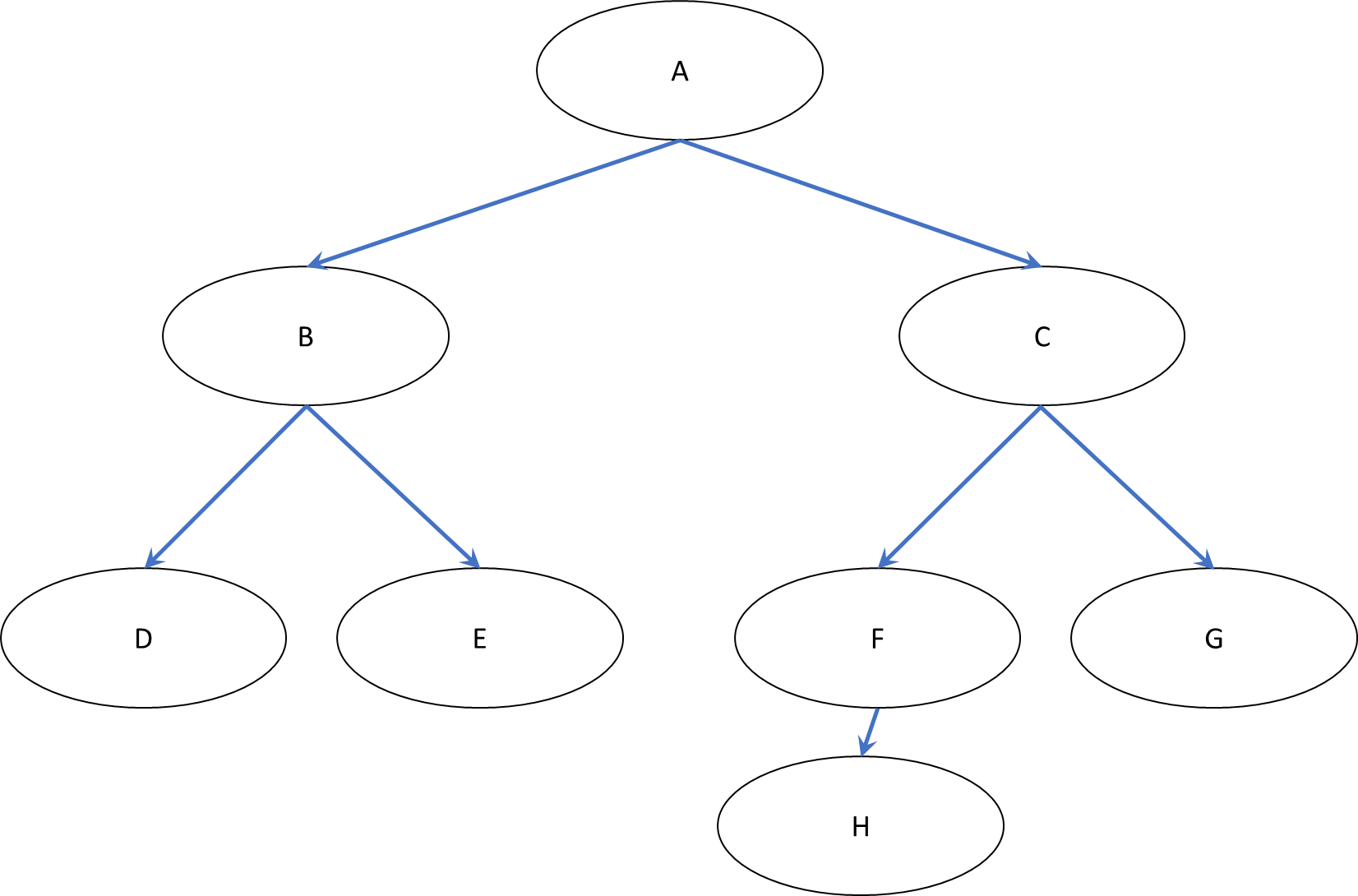


Figure 1

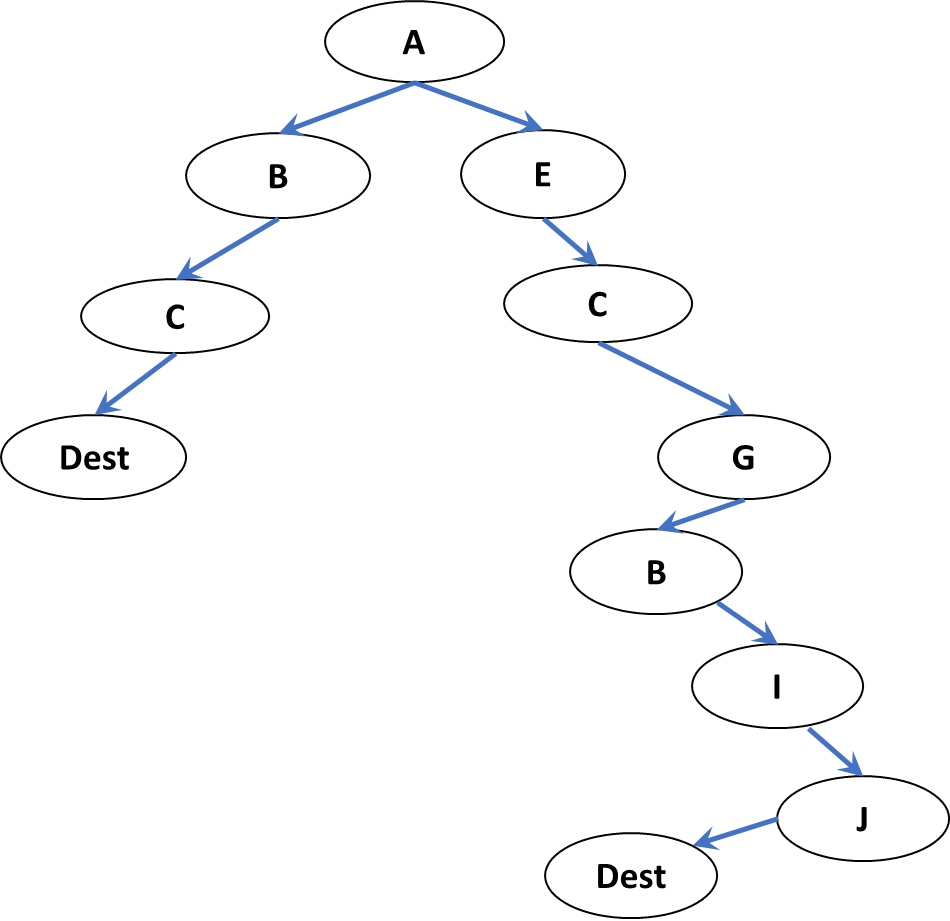


Figure 2

So do we have any much simpler and efficient method? Yes, of course, the BFS (breadth-first-search) can be a substitutes of above DFS(depth-first-search).

Let us still use the Figure 1 as example. We can first process the 0th level element, then process the 1st level element, and next …….

To be specific, we first put A (the source string) into the queue. Then let us see what the property of the generated tree is.

1. First, on the same path there should be no same string. This has been described before, which is to avoid dead loop. For example, A, C, G should be all different.
2. Secondly, strings in different depth should be different. For example, B should be different with D, E, F and G, and C should also be different with D, E, F and G. Actually, you can see this second property includes property (1)

Let us assume strings in different depth can be the same. See figure 3. Assume we can get one optimal path from A to Dest : ABDCKDest. We can see this path includes a level 1 string C (A is at level 0). However, this is definitely not an optimal path because we can move K and Dest under Level 1’s C and get a much more optimal path ACKDest. Thus, we can get that if there is one optimal path through A->Dest, then this path would not include any lower level strings.

The strings on the same level can be the same, and we can just process one of them because they will have the same optimal path solvation. For example, in Figure 3, M and N could be the same string, and their sub trees should be exactly the same.

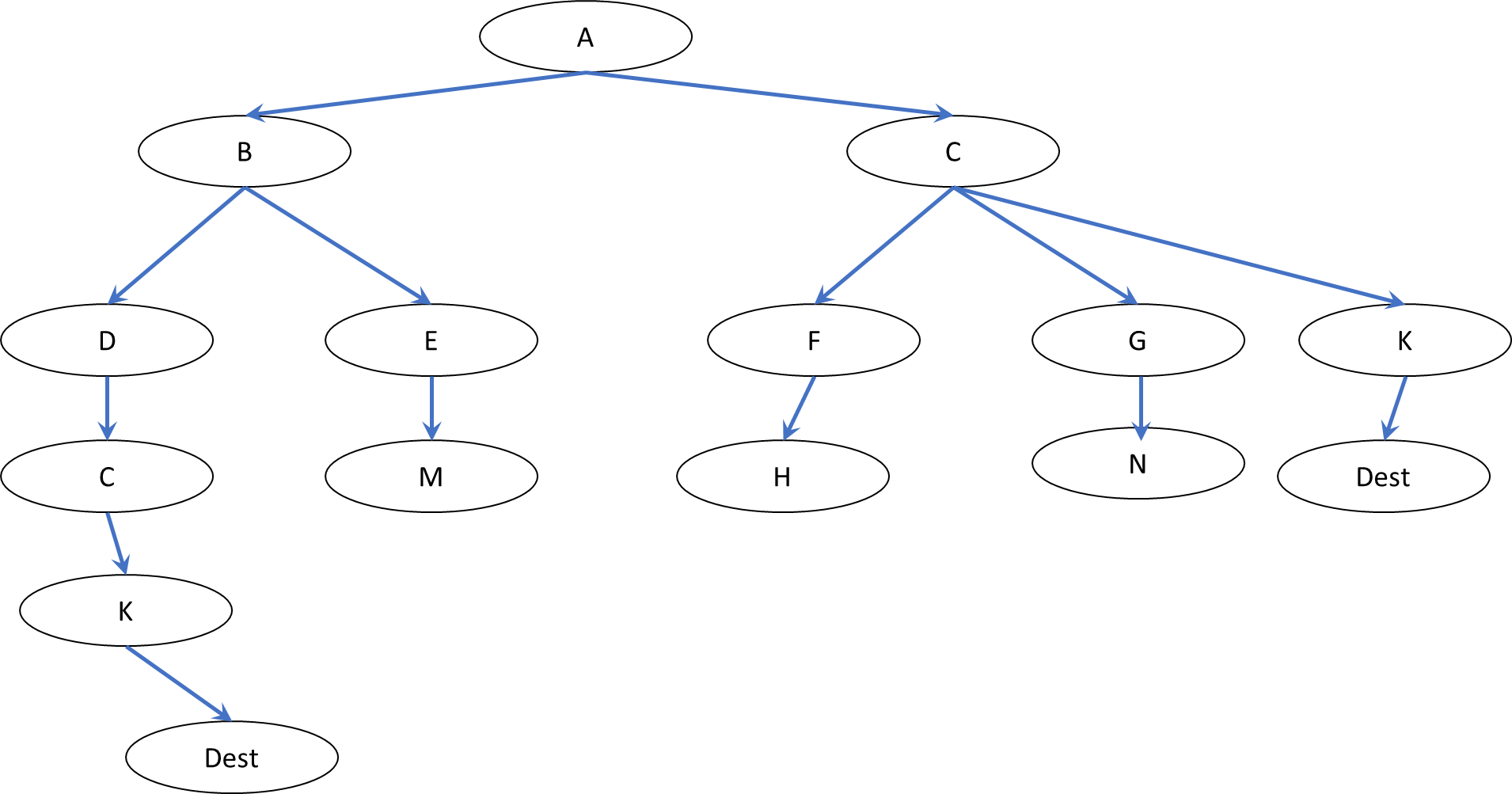


Figure 3

Based on the above property, we can get a BFS solution:

Prepare the following data structures:

* curLevelQueue<string>

stores the current level (which we are processing) strings

* nextLevelQueue<string>

stores the next level strings (which we will process after processing all current level strings)

* unordered\_map<string, vector<string>>

The key is a string, the value is a vector of strings. The vector records the key comes from which upper level strings. For example, if M = N in figure 3, then unordered\_map[M] = {E, G}. This is used to backtrace the path from the root. This is also used to "pruning" to reduce repeat computation. For example, we have described M and N should have the same sub trees if M = N. So we can just put M into the level 3's queue, no need to put N into the same queue again as they have the same results. This unordered\_map can be used to check whether we generates the same strings in next level when processing current level strings.

Take figure 3 as example, assume we are processing level 2. When processing E, we generates transformed string M, so make unordered\_map[M] = {E} and put M into nextLevelQueue. Later we process G, and generates transformed string N (N = M). We checked the unordered\_map and found we already have N, so push G into the unordered\_map[N(=M)] (thus unordered\_map[N(=M)] = {E, G} now) and do not put N into nextLevelQueue.

After processing all strings in level 2, we switch the nextLevelQueue and the curLevelQueue and repeat the same process.