Efficient Code

Abhishek Dey

Bidirectional Search : Two-End BFS

DECEMBER 13, 2017JANUARY 3, 2018 / EFFICIENTCODEBLOG

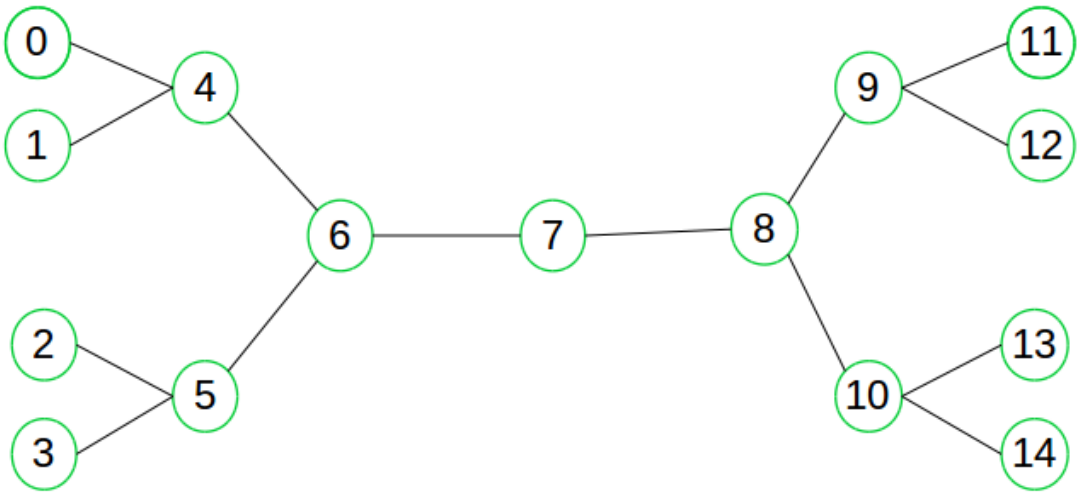
Bidirectional search is a graph search algorithm which find smallest path form source to goal vertex. It runs two simultaneous search –

1. Forward search form source/initial vertex toward goal vertex
2. Backward search form goal/target vertex toward source vertex

Bidirectional search replaces single search graph(which is likely to grow exponentially) with two smaller sub graphs – one starting from initial vertex and other starting from goal vertex. **The search terminates when two graphs intersect.**

> *Bidirectional Search* using B*readth First Search* which is also known as *Two-End BFS* gives the **shortest path** between the *source* and the *target*.

Consider following simple example-



Suppose we want to find if there exists a path from vertex 0 to vertex 14. Here we can execute two searches, one from vertex 0 and other from vertex 14. When both forward and backward search meet at vertex 7, we know that we have found a path from node 0 to 14 and search can be terminated now. We

can clearly see that we have successfully avoided unnecessary exploration.

Why bidirectional approach?

Because in many cases it is faster, it dramatically reduce the amount of required exploration. Suppose if branching factor of tree is **b** and distance of goal vertex from source is **d**, then the normal BFS/DFS searching complexity would be **O(b^d)**. On the other hand, if we execute two search operation then the complexity would be **O(b^{d/2})** for each search and total complexity would be **O(b^{d/2}+b^{d/2})** which is far less than **O(b^d)**.

When to use bidirectional approach?

We can consider bidirectional approach when-

1. Both initial and goal states are unique and completely defined.
2. The branching factor is exactly the same in both directions.

Performance measures

- **Completeness** : Bidirectional search is complete if BFS is used in both searches.
- **Optimality** : It is optimal if BFS is used for search and paths have uniform cost.
- **Time and Space Complexity** : Time and space complexity is **O(b^{d/2})**

Algorithm

Below is the pseudocode of the Bidirectional Search:

```
BIDIRECTIONAL_SEARCH
1   Q_I.Insert(x_I) and mark x_I as visited
2   Q_G.Insert(x_G) and mark x_G as visited
3   while Q_I not empty and Q_G not empty do
4       if Q_I not empty
5           x ← Q_I.GetFirst()
6           if x = x_G or x ∈ Q_G
7               return SUCCESS
8           forall u ∈ U(x)
9               x' ← f(x, u)
10              if x' not visited
11                  Mark x' as visited
12                  Q_I.Insert(x')
13              else
14                  Resolve duplicate x'
15      if Q_G not empty
16          x' ← Q_G.GetFirst()
17          if x' = x_I or x' ∈ Q_I
18              return SUCCESS
19          forall u^-1 ∈ U^-1(x')
20              x ← f^-1(x', u^-1)
21              if x not visited
22                  Mark x as visited
23                  Q_G.Insert(x)
24              else
25                  Resolve duplicate x
26  return FAILURE
```

Implementation

Let's see a naive C++ implementation of the ***bidirectional search*** using **BFS**:

```
// Method for bidirectional searching
int Graph::biDirSearch( int s,  int t)
{
     // boolean array for BFS started from
     // source and target(front and backward BFS)
     // for keeping track on visited nodes
     bool s_visited[V], t_visited[V];
     // Keep track on parents of nodes
     // for front and backward search
     int s_parent[V], t_parent[V];
     // queue for front and backward search
     list< int > s_queue, t_queue;
```

```
    int  intersectNode = -1;
    // necessary initialization
    for ( int  i=0; i<V; i++)
    {
        s_visited[i] =  false ;
        t_visited[i] =  false ;
    }
    s_queue.push_back(s);
    s_visited[s] =  true ;
    // parent of source is set to -1
    s_parent[s]=-1;
    t_queue.push_back(t);
    t_visited[t] =  true ;
    // parent of target is set to -1
    t_parent[t] = -1;
    while  (!s_queue.empty() && !t_queue.empty())
    {
        // Do BFS from source and target vertices
        BFS(&s_queue, s_visited, s_parent);
        BFS(&t_queue, t_visited, t_parent);
        // check for intersecting vertex
        intersectNode = isIntersecting(s_visited, t_visited);
        // If intersecting vertex is found
        // that means there exist a path
        if (intersectNode != -1)
        {
            cout <<  "Path exist between "  << s <<  " and "
                 << t <<  "\n" ;
            cout <<  "Intersection at: "  << intersectNode <<  "\n" ;
            // print the path and exit the program
            printPath(s_parent, t_parent, s, t, intersectNode);
            exit (0);
        }
    }
    return  -1;
}
```

- **Now let's have a look at the BFS() and isIntersecting() methods:**

```
// Method for Breadth First Search
void  Graph::BFS(list< int > *queue,  bool  *visited,
                                      int  *parent)
{
    int  current = queue->front();
    queue->pop_front();
    list< int >::iterator i;
    for  (i=adj[current].begin();i != adj[current].end();i++)
    {
```

```
            // If adjacent vertex is not visited earlier
            // mark it visited by assigning true value
            if  (!visited[*i])
            {
                // set current as parent of this vertex
                parent[*i] = current;
                // Mark this vertex visited
                visited[*i] =  true ;
                // Push to the end of queue
                queue->push_back(*i);
            }
        }
};
// check for intersecting vertex
int  Graph::isIntersecting( bool  *s_visited,  bool  *t_visited)
{
        int  intersectNode = -1;
        for ( int  i=0;i<V;i++)
        {
            // if a vertex is visited by both front
            // and back BFS search return that node
            // else return -1
            if (s_visited[i] && t_visited[i])
                return  i;
        }
        return  -1;
};
```

Another Implementation

In this approach we basically do BFS from the source in forward direction and from the target in backward direction, alternatively, step-by-step. We don't complete any of these two BFS wholly. Rather we incrementally approach towards each other from both direction and meet in at a intersection point (i.e, intersecting node) in the middle. This would become even clearer when you look the code shortly.

We take two queues: ***sourceQueue*** for BFS in *forward direction* from source to target and **targetQueue** which is used to do the BFS from the target towards the source in *backward direction*. We try to alternate between the two queues: sourceQueue and targetQueue; **basically in every iteration we choose the smaller queue for the next iteration** for the processing which effectively helps in alternating between the two queues only when the swapping between the two queues is profitable.

- This helps in the following way:

As we go deeper into a graph the number of edges can grow exponentially. Since we are approaching in a balanced way, selecting the queue which has smaller number of nodes for the next iteration, we are avoiding processing a large number of edges and nodes by trying to having the intersection point somewhere in the middle.

Since we are processing both the target and source queue we are not going to much depth from any direction, either in forward direction (i.e, while processing source queue) or in backward direction (i.e, target queue which searches from target to source in backward manner).

Since we are starting BFS from source and target and meeting somewhere in the middle we are processing moderate number of nodes from each direction since we are not traversing the graph from starting point to all the way bottom to the leaf nodes (if we imagine the graph as tree for the time being) as we stop search in each direction somewhere in the middle which helps us avoiding exponentially increasing number of nodes towards the bottom (remember BFS searches all nodes in each level).

> In the naive implementation since we were doing two complete BFS, one in forward direction from source to target, and another in backward direction from target to source, we were **unnecessarily** traversing O(b^d)nodes with all the exponentially growing huge number of nodes. In this more efficient implementation we are traversing just O(b^{d/2}) nodes in each direction (i.e, in each of forward and backward BFS), which is a huge improvement in the performance, since in both the direction we are avoiding reaching the depth with huge number of leaves, we are stopping at the middle depth of the graph.

We will go into the implementation of this efficient approach by solving an interesting using this **efficient Two-End BFS** algorithm.

Problem Statement:
Word Ladder: Given two words (*beginWord* and *endWord*), and a dictionary's word list, find the length of shortest transformation sequence from *beginWord* to *endWord*, such that:

1. Only one letter can be changed at a time.
2. Each transformed word must exist in the word list. Note that *beginWord* is *not* a transformed word.

For example,

Given:
beginWord = `"hit"`
endWord = `"cog"`
wordList = `["hot","dot","dog","lot","log","cog"]`

As one shortest transformation is `"hit" -> "hot" -> "dot" -> "dog" -> "cog"`,
return its length `5`.

Let's use Java for this implementation.

Solution # 1:

```
public int ladderLength(
        String beginWord,
        String endWord,
        List wordList
) {

 if (!wordList.contains(endWord)) {
 return 0;
 }
 Set dict = new HashSet(wordList);
 Set sourceQueue = new HashSet();
 Set targetQueue = new HashSet();
 Set visited = new HashSet();

 sourceQueue.add(beginWord);
 targetQueue.add(endWord);

 for (int len = 2; !sourceQueue.isEmpty(); len++) {

 Set transformedWordList = new HashSet();

 for (String w : sourceQueue) {

 for (int j = 0; j < w.length(); j++) {

 char[] ch = w.toCharArray();

 for (char c = 'a'; c <= 'z'; c++) {

 // continue when you get the parent word back
 if (c == w.charAt(j)) continue;
 ch[j] = c;
 String transformedWord = String.valueOf(ch);

 if (targetQueue.contains(transformedWord)) {
 return len; // meet from two ends; so return
 }

 if (dict.contains(transformedWord) && visited.add(transformedWord)) {
 transformedWordList.add(transformedWord);
 }
 }
 }
 }

 // always traverse the smaller one
 // BASICALLY IN EVERY ITERATION WE ARE TRYING
 // TO ALTERNATE BETWEEN SOURCE QUEUE & TARGET QUEUE
```

```
    sourceQueue = (transformedWordList.size() < targetQueue.size())
            ? transformedWordList
            : targetQueue;

    targetQueue = (sourceQueue == transformedWordList)
            ? targetQueue
            : transformedWordList;

    }

    return 0;
    }
```

Solution # 2:

```
public int ladderLength(
        String beginWord,
        String endWord,
        List wordList
) {

 if (!wordList.contains(endWord)) {
 return 0;
 }

 int len = 1;

 Set beginSet = new HashSet();
 Set endSet = new HashSet();
 Set visited = new HashSet();

 beginSet.add(beginWord);
 endSet.add(endWord);
 visited.add(beginWord);
 visited.add(endWord);

 while (!beginSet.isEmpty()) { // not including targetQueue here,
 // since we are always assigning the smaller size queue to sourcesQueue
 // at the end of each iteration
 // This will help to terminate earlier

 Set next = new HashSet();

 len++;

 for (String str : beginSet) {

 char[] ch = str.toCharArray();

 // construct all possible words
 for (int i = 0; i < str.length(); i++) {

 for (char c = 'a'; c <= 'z'; c++) {

 if (c == ch[i]) continue; // continue when you get the parent word back
 ch[i] = c;
 String word = String.valueOf(ch);

 if (endSet.contains(word)) { // INTERSECTION POINT
 return len;
 }

 if (wordList.contains(word) && !visited.contains(word)) {
```

```
visited.add(word);
next.add(word);
}

ch[i] = str.charAt(i);
}
}
}

if (next.size() < endSet.size()) {
beginSet = next;
} else {
beginSet = endSet;
endSet = next;
}

}
return 0;
}
```

Solution # 3:

```
public int ladderLength(
        String beginWord,
        String endWord,
        List wordList
) {

 if (!wordList.contains(endWord)) {
 return 0;
 }int pathLength = 2;

 Set start = new HashSet();
 Set end = new HashSet();
 start.add(beginWord);
 end.add(endWord);
 wordList.remove(beginWord);
 wordList.remove(endWord);

 while(!start.isEmpty()){
 if(start.size() > end.size()){
 Set temp = start;
 start = end;
 end = temp;
 }
 Set next = new HashSet();
 for(String cur : start){
 char[] strArray = cur.toCharArray();
 for(int i = 0; i < strArray.length;i++){
 char old = strArray[i];
 for(char c = 'a';c <= 'z';c++){
 strArray[i] = c;
 String str = String.valueOf(strArray);
 if(end.contains(str)){
 return pathLength;
 }
 if(wordList.contains(str)){
 next.add(str);
 wordList.remove(str);
 }
 }
 strArray[i] = old;
 }
 }
 start = next;
 pathLength++;
 }
 return 0;
 }
```