BFS算法框架

BFS找到的路径一定是最短的,代价是空间复杂度比DFS大很多。 本质上BFS问题： **在一幅图中找到从起点`start`到终点`target`的最近距离。** BFS框架：

```
// 计算起点start到终点target的最短距离
int BFS(Node start, Node end) {
  std::queue<Node> queue; // 核心数据结构
  std::set<Node> visited; // 避免走回头路

  queue.push(start); // 将起点加入队列
  visited[start] = true; // 标记已访问
  int step = 0; // 记录扩散的步数
  while(!queue.empty()) {
    int sz = queue.size();
    // 将队列的所有节点向四周扩散
    for(int i = 0; i < sz;i++) {
      Node cur = queue.poll();
      // 判断是否到达终点
      if(cur is target) {
        return step;
      }
      // 将cur 相邻的节点加入队列
      for(Node x: cur.adj()) {
        if(x is not visited) {
          queue.push(x);
          visited[x] = true;
        }
      }
    }
    step++; // 更新步数
  }
}
```

二叉树最小深度

```
class Solution {
public:
  int minDepth(TreeNode *root) {
    if (!root) {
      return 0;
    }
    std::deque<TreeNode *> que;
    que.push_back(root);
    int depth = 1;
    while (que.size()) {
      int size = que.size();
      for (int i = 0; i < size; ++i) {
        TreeNode *curr = que.front();
```

```
        que.pop_front();
        if (curr->left == nullptr && curr->right == nullptr) {
          return depth;
        }
        if (curr->left) {
          que.push_back(curr->left);
        }

        if (curr->right) {
          que.push_back(curr->right);
        }
      }
      depth++;
    }

    return 0;
  }
};
```

解开密码锁的最少次数

```
class Solution {
public:
  int openLock(vector<string>& deadends, string target) {
    std::deque<std::string> que;
    int                     step = 0;
    que.push_back("0000");
    std::unordered_map<std::string, int> dead;
    for (auto item : deadends) {
      dead[item] = 1;
    }
    std::unordered_map<std::string, bool> visited;
    visited["0000"] = true;

    while (que.size()) {
      int size = que.size();

      for (int i = 0; i < size; i++) {
        std::string curr = que.front();
        que.pop_front();
        // 是否为dead
        if (dead.count(curr)) {
          continue;
        }
        if (curr == target) {
          return step;
        }

        for (int i = 0; i < 4; i++) {
          std::string up = plusOne(curr, i);
          if (!visited.count(up)) {
```

```
            que.push_back(up);
            visited[up] = true;
          }

          std::string down = minusOne(curr, i);
          if (!visited.count(down)) {
            que.push_back(down);
            visited[down] = true;
          }
        }
      }

      step++;
    }

    return -1;
  }

private:
  std::string plusOne(std::string str, int j) {
    if (str[j] == '9') {
      str[j] = '0';
    } else {
      str[j] += 1;
    }
    return str;
  }

  std::string minusOne(std::string str, int j) {
    if (str[j] == '0') {
      str[j] = '9';
    } else {
      str[j] -= 1;
    }

    return str;
  }
};
```

双向BFS优化：

```
#include <iostream>
#include <queue>
#include <string>
#include <unordered_set>
#include <vector>

// @lc code=start
class Solution {
public:
  int openLock(std::vector<std::string> &deadends, std::string target) {
    std::unordered_set<std::string> que;
```

```
    std::unordered_set<std::string> que2;  // 双向BFS
    std::unordered_set<std::string> set;
    std::unordered_set<std::string> visited;  // 剪枝使用
    for (auto &str : deadends) {
      set.insert(str);
    }

    if (target == "0000") {  // 处理target与start相同的情况
      return 0;
    }

    que.insert("0000");
    visited.insert("0000");
    visited.insert(target);
    que2.insert(target);
    int step = 1;  // 因为target也加入了队列，所以从1开始

    while (que.size() && que2.size()) {
      std::unordered_set<std::string> tmp;

      if (que.size() > que2.size()) {
        que.swap(que2);
      }

      for (auto iter = que.begin(); iter != que.end(); iter++) {
        std::string item = *iter;

        if (set.find(item) != set.end()) {
          continue;
        }

        if (que2.find(item) != que2.end()) {
          return step;
        }

        for (int i = 0; i < 4; i++) {
          std::string astr = addOne(item, i);
          if (visited.find(astr) == visited.end() &&
              set.find(astr) == set.end() && que2.find(astr) == que2.end())
{
            tmp.insert(astr);
            visited.insert(astr);
          }

          if (que2.find(astr) != que2.end()) {
            return step;
          }

          std::string mstr = minusOne(item, i);
          if (visited.find(mstr) == visited.end() &&
              set.find(mstr) == set.end() && que2.find(mstr) == que2.end())
{
            tmp.insert(mstr);
            visited.insert(mstr);
```

```
            }

            if (que2.find(mstr) != que2.end()) {
              return step;
            }
          }
        }

        step++;
        que.swap(que2);
        que2.swap(tmp);
      }

      return -1;
    }

  private:
    std::string addOne(std::string str, int i) {
      if (str[i] == '9') {
        str[i] = '0';
      } else {
        str[i] += 1;
      }

      return str;
    }

    std::string minusOne(std::string str, int i) {
      if (str[i] == '0') {
        str[i] = '9';
      } else {
        str[i] -= 1;
      }

      return str;
    }
  };
```

双向BFS优化

传统BFS是从start向四周扩散到target终止; 双向BFS是start和target同时开始扩散，当两边有交集时终止。 双向优化框架：

```
int twoBFS(std::vector<T> &depends,T start ,T target) {
  std::unordered_set<T> set;

  for(auto s:depends) {
    set.insert(s);
  }

  // 使用集合加快元素判断
```

```
    std::unordered_set<T> q1;
    std::unordered_set<T> q2;
    std::unordered_set<T> visited;

    int step = 0;

    q1.insert(start);
    q2.insert(target);

    while(!q1.empty() && !q2.empty()) {
      // 哈希集合在遍历时不能修改，用temp扩散结果
      std::unordered_set<T> temp;

      // 优化，判断q1和q2的大小，每次从最小的集合开始
      if(q1.size() > q2.size()) {
        temp = q1;
        q1 = q2;
        q2 = temp;
      }

      // 将q1的所有节点向四周扩散
      for(auto s : q1) {
        // 判断是否到达终点
        if(set.find(s) != set.end()) {
          continue;
        }

        if(q2.find(s) != q2.end()) {
          return step;
        }

        visited.insert(s);

        // 将每个节点未遍历节点加入集合
        temp.insert(next);

      }

      // 增加步数
      step++;
      // 此时temp 相当于q1
      // 交换q1,q2，下一次循环扩散q2
      q1 = q2;
      q2 = temp;
    }

    return -1;
  }
```