拓扑排序

给定一个课程表判断是否能够完成所有课程

1. 给定的为一个二维数组，需要将二维数组转换成有向图；
2. 是否能够完成所有课程，只需要判断有向图中是否存在环即可。

```
class Solution {
public:
  bool canFinish(int numCourses, std::vector<std::vector<int>>&
prerequisites) {
    std::unordered_map<int, std::list<int>> graph =
        buildGraph(numCourses, prerequisites);
    _visited = std::vector<bool>(numCourses, false);
    _onpath  = std::vector<bool>(numCourses, false);
    hasCycle = false;

    for (int i = 0; i < numCourses; i++) {
      trave(graph, i);
    }

    return !hasCycle;
  }

private:
  void trave(std::unordered_map<int, std::list<int>> graph, int s) {
    if (_onpath[s]) {
      hasCycle = true;
    }

    // 出现环
    if (_visited[s] || hasCycle) {
      return;
    }

    _visited[s] = true;
    _onpath[s]  = true;

    for (auto item : graph[s]) {
      trave(graph, item);
    }

    _onpath[s] = false;
  }

  std::unordered_map<int, std::list<int>> buildGraph(
      int                                numCourse,
      std::vector<std::vector<int>>& prerequisited) {
    std::unordered_map<int, std::list<int>> graph(numCourse + 1);

    for (int i = 1; i < numCourse; i++) {
```

```
      graph[i] = std::list<int>();
    }

    for (auto item : prerequisited) {
      int from = item[0];
      int to   = item[1];

      graph[from].push_back(to);
    }

    return graph;
  }

  std::vector<bool> _visited;  // 防止重复遍历同一个节点
  std::vector<bool> _onpath;   // 记录一次
  bool              hasCycle;  // 记录是否存在环
};
```

上述方法为DFS算法，在量大的情况下，容易导致超时。

BFS算法：

```
class Solution {
public:
  bool canFinish(int numCourses, std::vector<std::vector<int>>&
prerequisites) {
    std::unordered_map<int, std::list<int>> graph =
        buildGraph(numCourses, prerequisites);
    std::vector<int> indegree(numCourses, 0);

    for (int i = 0; i < numCourses; i++) {
      for (auto item : graph[i]) {
        indegree[item]++;
      }
    }

    std::list<int> list;
    // 所有入度为0的节点放入队列
    for (int i = 0; i < numCourses; i++) {
      if (0 == indegree[i]) {
        list.push_back(i);
      }
    }

    std::vector<int> order(numCourses, 0); // 拓扑排序的结果
    int              index = 0;
    // BFS遍历
    while (list.size()) {
      int curr = list.front();
      list.pop_front();

      order[index++] = curr;
```

```
      for (auto item : graph[curr]) {
        indegree[item]--;
        // 入度为0加入队列
        if (0 == indegree[item]) {
          list.push_back(item);
        }
      }
    }

    // 遍历完成结束之后，判断排序的数组长度是否等于课程数，相等的话，可以完成，不等的话
完不成
    return index == numCourses;
  }

private:
  std::unordered_map<int, std::list<int>> buildGraph(
      int                                numCourse,
      std::vector<std::vector<int>>& prerequisited) {
    std::unordered_map<int, std::list<int>> graph(numCourse + 1);

    for (int i = 1; i < numCourse; i++) {
      graph[i] = std::list<int>();
    }

    for (auto item : prerequisited) {
      int from = item[0];
      int to   = item[1];

      graph[from].push_back(to);
    }

    return graph;
  }
};
```

进阶返回所有课程的学习顺序

```
class Solution {
public:
  std::vector<int> findOrder(int                                numCourses,
                             std::vector<std::vector<int>>& prerequisites)
{
    std::vector<std::vector<int>> graph = buildGraph(numCourses,
prerequisites);
    std::vector<int>              indegree(numCourses);
    std::deque<int>               que;

    for (int i = 0; i < numCourses; i++) {
      for (auto item : graph[i]) {
        indegree[item]++;
      }
```

```
    }

    for (int i = 0; i < numCourses; i++) {
      if (0 == indegree[i]) {
        que.push_back(i);
      }
    }

    std::vector<int> order(numCourses, 0);
    int              index = 0;

    while (que.size()) {
      int v = que.front();
      que.pop_front();

      order[index++] = v;

      for (auto item : graph[v]) {
        indegree[item]--;
        if (0 == indegree[item]) {
          que.push_back(item);
        }
      }
    }

    if (index != numCourses) {
      return std::vector<int>();
    } else {
      std::reverse(order.begin(), order.end());
      return order;
    }
  }

private:
  std::vector<std::vector<int>> buildGraph(
      int                           numCourses,
      std::vector<std::vector<int>>& prerequisites) {
    std::vector<std::vector<int>> graph(numCourses, std::vector<int>());

    for (int i = 0; i < prerequisites.size(); i++) {
      int from = prerequisites[i][0];
      int to   = prerequisites[i][1];
      graph[from].push_back(to);
    }

    return graph;
  }
};
```