《人工智能》课程设计报告

选题名称：图搜索算法对比研究

学号：M201973167

姓名：高也

**报告题目：图搜索算法对比研究**

（1）编程实现宽度优先、等代价、深度优先、有序搜索算法；

（2）要求用可视化界面演示算法执行过程，能单步执行，也能连续执行，能画出搜索树，能展示OPEN表和CLOSED表的动态变化过程；

（3）可自定义搜索图，通过实验验证几种图搜索算法的性能。

报告要求：

1、摘要

2、问题描述、知识表示

3、说明采用何种算法实现（或多个算法融合）

4、算法的原理、步骤、过程

5、开发工具不限

6、编写代码简单实现，截图说明。

7、小结及展望

摘要

本报告中涉及了宽度优先搜索算法、深度优先搜索算法、等代价搜索算法、最佳优先搜索算法。对于每一种算法，我们分别介绍它们的原理，步骤和实现代码。

1. **宽度优先搜索(BFS)算法。**
   1. 宽度优先搜索(BFS)算法原理。

广度优先搜索算法（英语：Breadth-First-Search，缩写为BFS），又译作宽度优先搜索，或横向优先搜索，是一种图形搜索算法。简单的说，BFS是从根节点开始，沿着树的宽度遍历树的节点。如果所有节点均被访问，则算法中止。广度优先搜索的实现一般采用OPEN表和CLOSED表。

BFS是一种盲目搜索法，目的是系统地展开并检查图中的所有节点，以找寻结果。换句话说，它并不考虑结果的可能地址，彻底地搜索整张图，直到找到结果为止。BFS并不使用经验法则算法。

从算法的观点，所有因为展开节点而得到的子节点都会被加进一个先进先出的队列中。一般的实现里，其邻居节点尚未被检验过的节点会被放置在一个被称为OPEN表中（例如队列或是链表），而被检验过的节点则被放置在被称为 CLOSED表中。

* 1. 宽度优先搜索(BFS)算法步骤。

Step 1，首先将根节点放入队列中。

Step 2，从队列中取出第一个节点，并检验它是否为目标。

如果找到目标，则结束搜索并回传结果。

否则将它所有尚未检验过的直接子节点加入队列中。

Step 3，若队列为空，表示整张图都检查过了——亦即图中没有欲搜索的目标。结束搜索并回传“找不到目标”。

Step 4，重复Step 2。

宽度优先搜索算法流程图如图1所示。

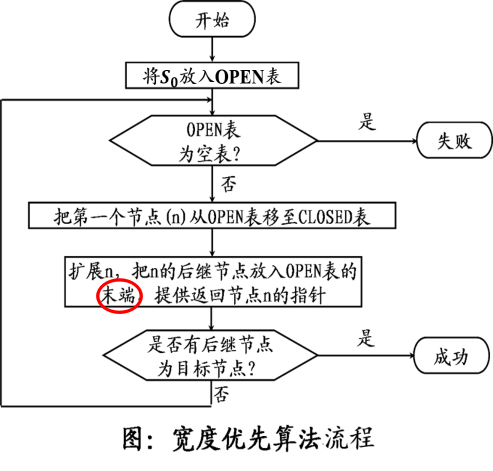


图1：宽度优先搜索算法流程。

* 1. 宽度优先搜索(BFS)算法实验。

我们使用图2所示的有向图，来验证宽度优先搜索算法。不妨设1为起始结点，5为目标结点。

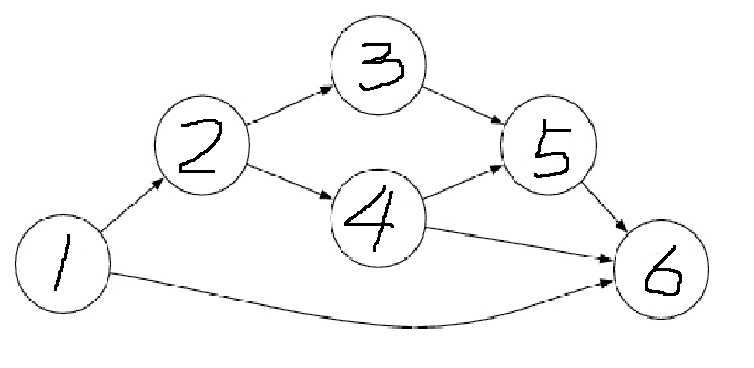


图2：实验示例图。

宽度优先搜索生成的搜索树如图3所示。

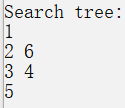


图3：宽度优先搜索生成的搜索树。

执行宽度优先搜索时，搜索树和OPEN表和CLOSED表的变化过程如图3所示。我们注意到，第二步时OPEN表变为空。由于这里不是OPEN表的检测点，所以程序并不会跳出，而是会继续执行下去。

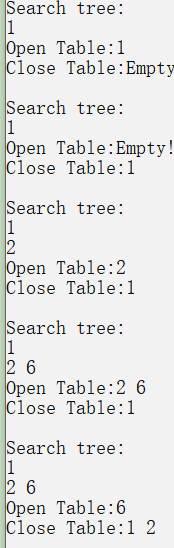
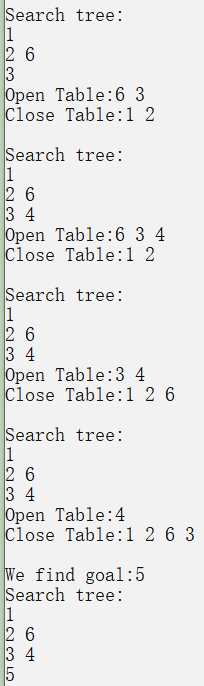
 

图4：BFS执行过程中，搜索树和OPEN表和CLOSED表的动态变化。

单步执行宽度优先搜索时，搜索树和OPEN表和CLOSED表的变化过程如图5所示。

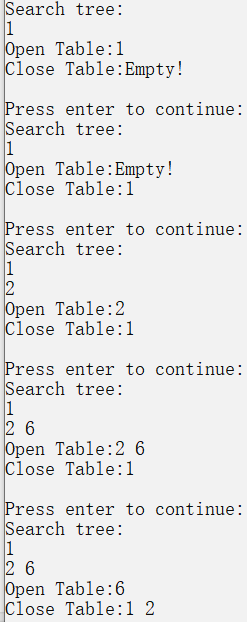
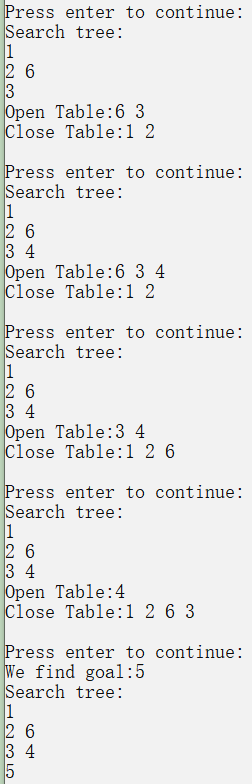
 

图5：BFS单步执行过程中，搜索树和OPEN表和CLOSED表的动态变化。

宽度优先搜索(BFS)的完整代码见附录5.1。

1. **深度优先搜索(DFS)算法。**
   1. 深度优先搜索(DFS)算法原理。

深度优先搜索算法（英语：Depth-First-Search，DFS）是一种用于遍历或搜索树或图的算法。沿着树的深度遍历树的节点，尽可能深的搜索树的分支。当节点v的所在边都己被探寻过，搜索将回溯到发现节点v的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。如果还存在未被发现的节点，则选择其中一个作为源节点并重复以上过程，整个进程反复进行直到所有节点都被访问为止。属于盲目搜索。

深度优先搜索是图论中的经典算法，利用深度优先搜索算法可以产生目标图的相应拓扑排序表，利用拓扑排序表可以方便的解决很多相关的图论问题，如最大路径问题等等。

* 1. 深度优先搜索(DFS)算法步骤。

Step 1，首先将根节点放入stack中。

Step 2，从stack中取出第一个节点，并检验它是否为目标。

如果找到目标，则结束搜寻并回传结果。

否则将它某一个尚未检验过的直接子节点加入stack中。

Step 3，重复Step 2。

Step 4，如果不存在未检测过的直接子节点。

将上一级节点加入stack中。

重复步骤2。

Step 5，重复Step 4。

Step 6，若stack为空，表示整张图都检查过了——亦即图中没有欲搜寻的目标。结束搜寻并回传“找不到目标”。

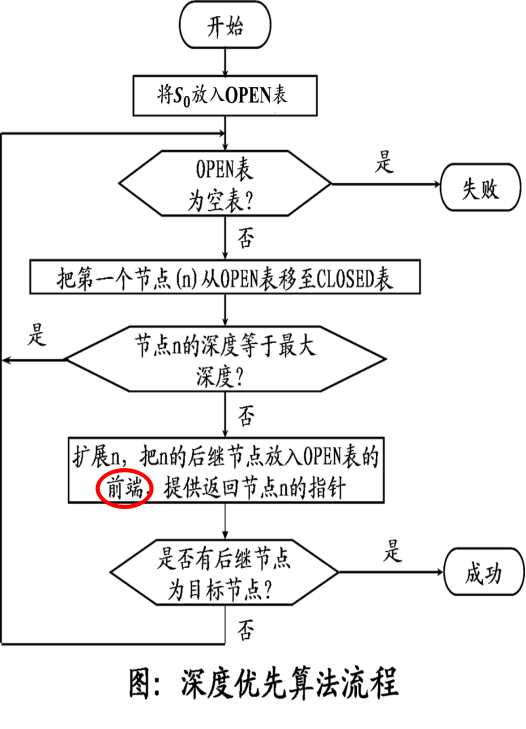


图5：深度优先算法流程。

* 1. 深度优先搜索(DFS)算法实验。

我们使用图5所示的有向图，来验证深度优先搜索算法。不妨设1为起始结点，4为目标结点。

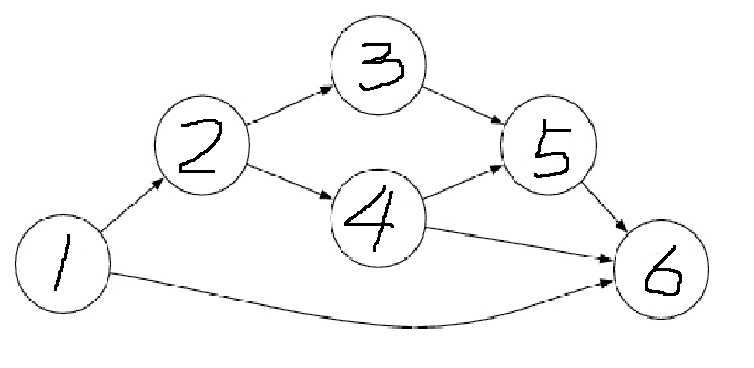


图6：实验示例图。

深度优先搜索生成的搜索树如图7所示。

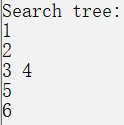


图7：深度优先搜索生成的搜索树。

执行深度优先搜索时，搜索树和OPEN表和CLOSED表的变化过程如图8所示。

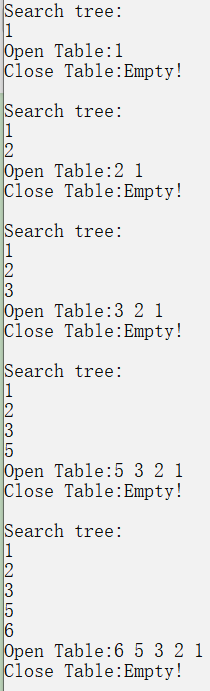
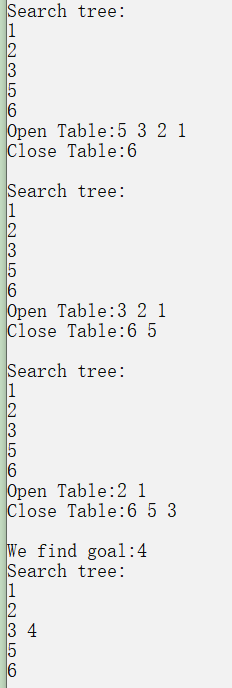
 

图8：DFS执行过程中，搜索树和OPEN表和CLOSED表的动态变化。

单步执行深度优先搜索时，搜索树和OPEN表和CLOSED表的变化过程如图6所示。

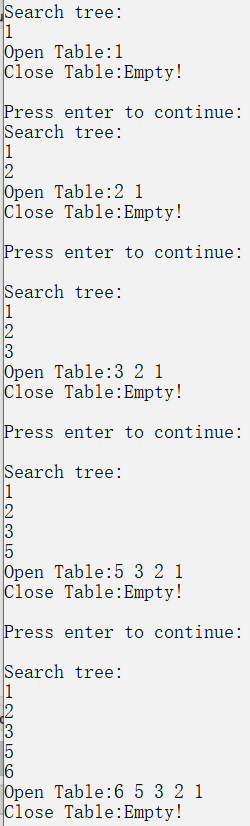
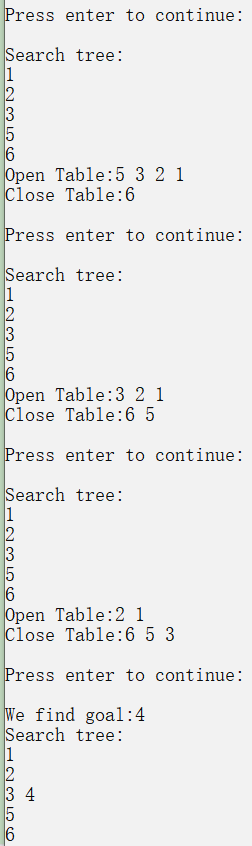
 

图9：DFS单步执行过程中，搜索树和OPEN表和CLOSED表的动态变化。

1. **等代价搜索算法。**
   1. 等代价搜索算法原理。

等代价搜索（UCS，uniform cost search）,又称一致性代价搜索。在前面的阐述中，我们都假设搜索树的连接的代价是相同的。但假设问题中的树做一些改变，使得每条连接上都有一定的代价参数，并在问题中给定了起始状态和目标状态。这时候，若我们使用宽度优先搜索，并解决寻找从起始状态到目标状态具有最小代价的路径问题，那么，这种被推广了的宽度优先搜索就称作等代价搜索。

* 1. 等代价搜索算法步骤。

为表述方便，我们做以下规定：把从节点i到它的后继节点j的连接弧线代价记为c(i,j)，把从起始节点S到任一节点i的路径代价记为g(i)

Step 1，把起始节点S放到未扩展节点表OPEN表中。如果起始节点为一目标节点，则求得一个解，结束；否则，令g(s) = 0，转step2。

Step 2，如果OPEN表为空，则表示没有解，结束；否则，转step3。

Step 3，从OPEN表中选择一个节点，使得g(i)最小。如果有几个节点都符合，若存在目标节点，则选择一个目标节点，结束；否则选择一个作为节点i。并把节点i从OPEN表移至CLOSED表中，转Step 4。

Step 4，如果节点i为目标节点，则求得一个解，结束；否则，转Step 5。

Step 5，扩展节点i。如果没有后继节点，转Step 2；否则，转Step 6。

Step 6，对于节点i的每个后继节点，计算g(i) = g(i) + c(i,j),并把所有后继节点j放在OPEN表中，提供返回到节点i的指针，转Step 2。

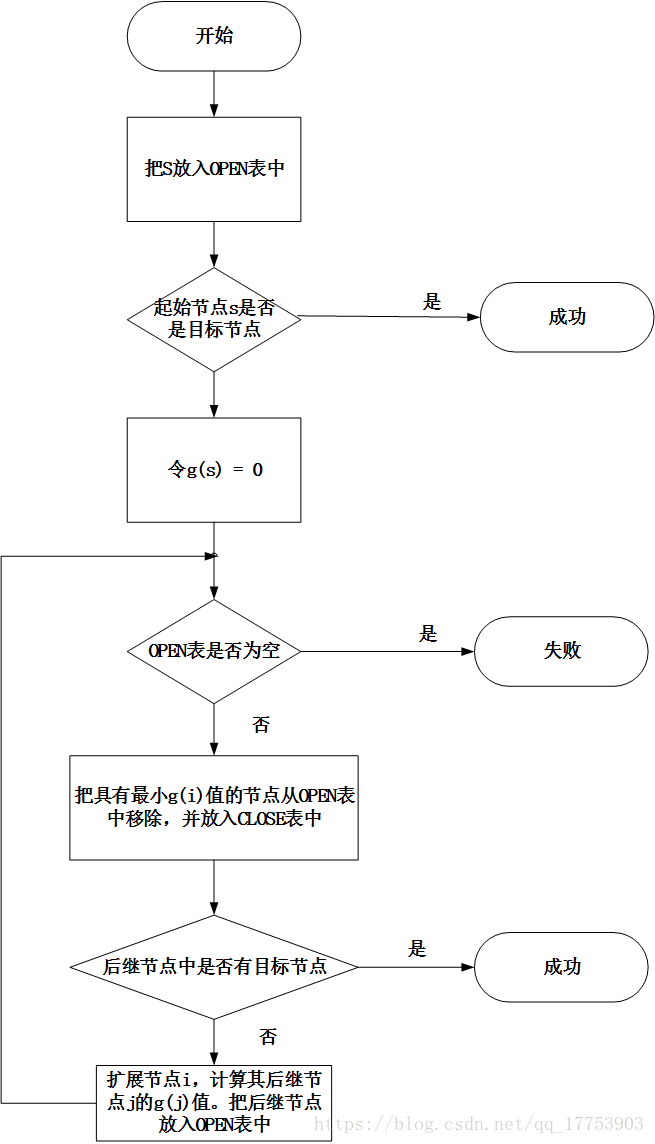


图7：深度优先算法流程。

1. **最佳优先搜索(有序搜索)算法。**
   1. 最佳优先搜索算法原理。

当我们在状态空间中搜索的时候，最简单的方法就是穷举，在之前文章提及到的广度优先搜索和深度优先搜索都属于穷举类型的搜索，这种搜索方法有一个很大的缺点，就是在状态空间十分大的时候效率非常的差，因为需要穷举的状态太多了。而启发式搜索就是对状态空间中的每个搜索的位置（如图中的节点）进行一个评估，然后选出最好的位置。而在启发估价中使用到的函数我们称之为启发估价函数。

启发估价函数：f(n) = g(n) + h(n)，其中n为现在所在的节点，g(n)为从起始点到点n的实际代价，h(n)为从点n到目标点的估价。

最佳优先搜索（Best First Search），是一种启发式搜索算法（Heuristic Algorithm）,我们也可以将它看做广度优先搜索算法的一种改进；最佳优先搜索算法在广度优先搜索的基础上，用启发估价函数对将要被遍历到的点进行估价，然后选择代价小的进行遍历，直到找到目标节点或者遍历完所有点，算法结束。

一致代价搜索和最佳优先搜索的区别如下：

1. 一致代价搜索是一种盲目搜索，而最佳优先搜索是一种启发式搜索。
2. 一致代价搜索无法处理启发函数，相当于f(n)=g(n)；而最佳优先搜索可以处理启发函数，相当于f(n)=g(n)+h(n)。
   1. 最佳优先搜索算法步骤。

要实现最佳优先搜索我们必须使用一个优先队列（priority queue）来实现，通常采用一个OPEN表和一个CLOSED表。OPEN表用来储存还没有遍历将要遍历的节点，而CLOSED表用来储存已经被遍历过的节点。

最佳优先搜索的过程可以被描述为：

Step 1，将根节点放入优先队列OPEN表中。

Step 2，从优先队列中取出优先级最高的节点X。

Step 3，根据节点X生成子节点Y:

(1) X的子节点Y不在OPEN表或者CLOSED表中，由估价函数计算出估价值，放入OPEN表中。

(2) X的子节点Y在OPEN表中，且估价值优于OPEN表中的子节点Y，将OPEN表中的子节点Y的估价值替换成新的估价值并按优先值排序。

(3) X的子节点Y在CLOSED表中，且估价值优于CLOSED表中的子节点Y，将CLOSED表中的子节点Y移除，并将子节点Y加入OPEN表。

Step 4，将节点X放入CLOSED表中。

Step 5，重复过程2,3,4直到目标节点找到，或者OPEN为空，程序结束。

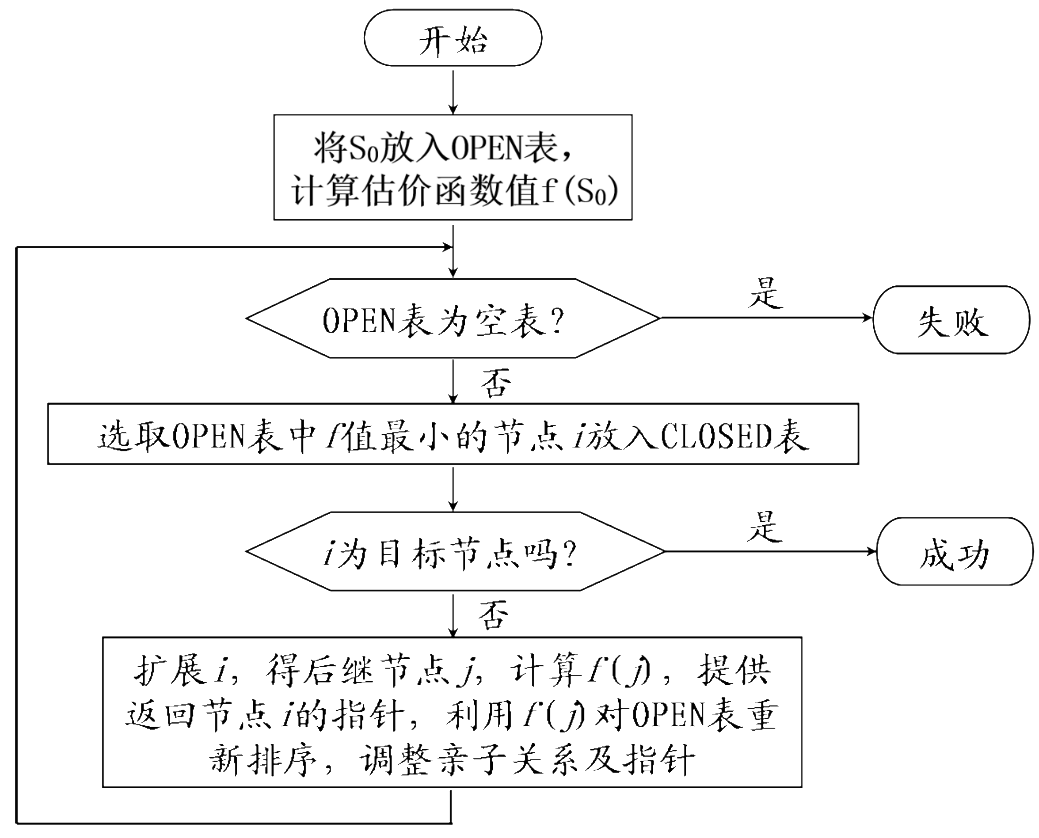


图6：最佳优先算法流程。

1. **附录。**
   1. 宽度优先搜索(BFS)算法代码。

我们使用C++来实现宽度优先搜索(BFS)代码。完整代码如表1所示。

|  |
| --- |
| #include<iostream>  #include<vector>  using namespace std;  int num\_v;  vector<bool> adjacency\_matrix;  vector<bool> visited;  vector<int> OpenTable;  vector<int> CloseTable;  struct Node  {  int key;  int level;  vector<int> child;  };  vector<Node> tree;  vector<int> result;  void print\_OpenTable();  void print\_CloseTable();  void print\_SearchTree();  bool test\_edge(int start, int end)  {  bool flag = false;  if(adjacency\_matrix[(start-1)\*num\_v+(end-1)]==true) flag = true;  return flag;  }  // BFS continuous execution;  bool BFS(int source, int goal)  {  visited[source-1] = true;  tree[source-1].level = 0;  // cout<<"Current Node:"<<i<<" "<<endl;  result.push\_back(source);  OpenTable.push\_back(source);    print\_SearchTree();  print\_OpenTable();  print\_CloseTable();  int current = -1;  while(!OpenTable.empty())  {  current = OpenTable.front();  OpenTable.erase(OpenTable.begin());  CloseTable.push\_back(current);    print\_SearchTree();  print\_OpenTable();  print\_CloseTable();  for(int j=1;j<=num\_v;j++)  {  if((test\_edge(current, j) == true)&&(visited[j-1]==false))  {  visited[j-1] = true;  // cout<<"Current Node:"<<j<<" "<<endl;  result.push\_back(j);  tree[current-1].child.push\_back(j);  tree[j-1].level = tree[current-1].level + 1;    if(j==goal)  {  cout<<"We find goal:"<<j<<endl;  print\_SearchTree();  return true;  }    OpenTable.push\_back(j);    print\_SearchTree();  print\_OpenTable();  print\_CloseTable();  }  }  }    cout<<"Not found!"<<endl;  return false;  }  // BFS one range;  bool BFS\_step(int source, int goal)  {  visited[source-1] = true;  tree[source-1].level = 0;  // cout<<"Current Node:"<<i<<" "<<endl;  result.push\_back(source);  OpenTable.push\_back(source);    print\_SearchTree();  print\_OpenTable();  print\_CloseTable();  cout<<"Press enter to continue: "<<endl;  cin.ignore();  int current = -1;  while(!OpenTable.empty())  {  current = OpenTable.front();  OpenTable.erase(OpenTable.begin());  CloseTable.push\_back(current);    print\_SearchTree();  print\_OpenTable();  print\_CloseTable();  cout<<"Press enter to continue: ";  cin.ignore();  for(int j=1;j<=num\_v;j++)  {  if((test\_edge(current, j) == true)&&(visited[j-1]==false))  {  visited[j-1] = true;  // cout<<"Current Node:"<<j<<" "<<endl;  result.push\_back(j);  tree[current-1].child.push\_back(j);  tree[j-1].level = tree[current-1].level + 1;    if(j==goal)  {  cout<<"We find goal:"<<j<<endl;  print\_SearchTree();  return true;  }    OpenTable.push\_back(j);    print\_SearchTree();  print\_OpenTable();  print\_CloseTable();  cout<<"Press enter to continue: ";  cin.ignore();  }  }  }    cout<<"Not found!"<<endl;  return false;  }  void print\_OpenTable()  {  cout<<"Open Table:";    if(OpenTable.size()==0)  {  cout<<"Empty!";  }  else  {  for(int i=0;i<OpenTable.size();i++)  {  cout<<OpenTable[i]<<" ";  }  }      cout<<endl;  }  void print\_CloseTable()  {  cout<<"Close Table:";    if(CloseTable.size()==0)  {  cout<<"Empty!";  }  else  {  for(int i=0;i<CloseTable.size();i++)  {  cout<<CloseTable[i]<<" ";  }  }  cout<<endl<<endl;  }  void print\_SearchTree()  {  cout<<"Search tree: "<<endl;  for(int i=0;i<result.size();i++)  {  if(i>0)  {  if(tree[result[i]-1].level > tree[result[i-1]-1].level)  {  cout<<endl;  }  }      cout<<result[i]<<" ";  }    cout<<endl;    return;  }  int main()  {  cout<<"Input number of node and number of edge in the first line, input each edge in the following line."<<endl;    int N; // number of vertices  cin>>N;  num\_v = N;  tree.resize(N);    for(int i=0;i<N;i++)  {  tree[i].level = 0;  }      int M; // number of edges  cin>>M;  // create adjacency matrix;  for(int i=0;i<N\*N;i++)  {  adjacency\_matrix.push\_back(false);  }  // save current amp in adjacency matrix;  for(int i=0;i<M;i++)  {  int start;  cin>>start;  int end;  cin>>end;  adjacency\_matrix[(start-1)\*N + (end-1)] = true;  }  // cout<<test\_edge(2, 1)<<endl;  // cout<<test\_edge(2, 4)<<endl;  // generate visited point;  visited.resize(N);  for(int i=0;i<N;i++)  {  visited[i] = false;  }  // BFS(1, 5); // BFS continuous execution;  BFS\_step(1, 5); //BFS step execution;    // print search tree;    cout<<endl;  for(int i=0;i<N;i++)  {  cout<<"Node:"<<i+1<<"; Level:"<<tree[i].level<<"; Child:";  for(int j=0;j<tree[i].child.size();j++)  {  cout<<tree[i].child[j]<<" ";  }  cout<<endl;  }    cout<<endl;  print\_SearchTree();  cout<<endl;    return 0;  }  /\*  6 8  1 2  1 6  2 3  2 4  3 5  4 5  4 6  5 6  \*/ |

表1：宽度优先搜索C++代码。