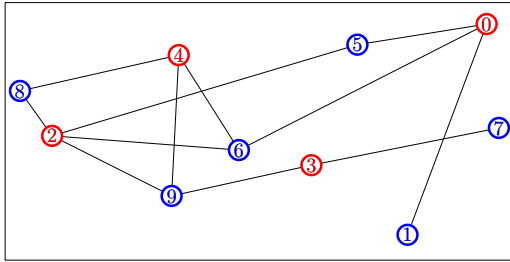


Lesson 20: Learn to Traverse Graphs



Breadth First Search, Depth First Search, Topological Sort

AICE1005

Algorithms and Analysis

1

Basic Graph Algorithms

- Graphs provide an abstraction for a huge number of real world processes: social networks, compute network, road networks, etc.
- Increasing applications focus on very large (sparse) graphs (usually implemented using an adjacency list)
- Require (near) linear time algorithms
- Basic building block are graph traversal algorithms
 - ★ Breadth First Search
 - ★ Depth First Search

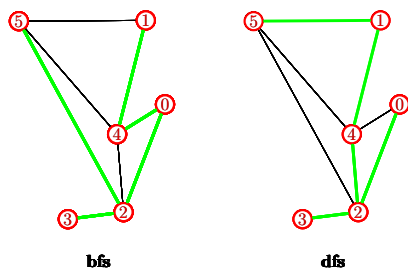
AICE1005

Algorithms and Analysis

3

Graph Traversal

- To traverse a graph we start at a (arbitrary) *root* vertex
- We follow edges to create a tree



AICE1005

Algorithms and Analysis

5

Applications of Breadth First Search

- Breadth first search can be used to find the shortest path from a source node to a destination node for an **unweighted** graph
 - ★ Run `bfs(graph, source)`
 - ★ Use parent information to find path from destination back to source
- BFS (as well as DFS) can be used to find connected components
 - ★ Use `processVertexEarly` to mark vertices connected to the current connected component
 - ★ Run `bfs` from all vertices that are not labelled

AICE1005

Algorithms and Analysis

7

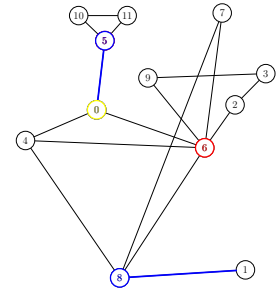
1. Breadth First Search

- BFS applications

2. Depth First Search

- DFS applications

3. Topological Sort



AICE1005

Algorithms and Analysis

2

Advanced Generic

- To make these algorithm general purpose (generic) we allow ourselves to call arbitrary functions to act on the vertices and edges at different points in the algorithm
- This introduces a new level of generics which makes the algorithms very powerful
- Increases the steepness of the learning curve to use these algorithms
- Once you get familiar with using these algorithms this level of generics starts to pay off
- Libraries which does this include Boost Graph Library and LEDA in C++; JDSL and JGraphT in Java

AICE1005

Algorithms and Analysis

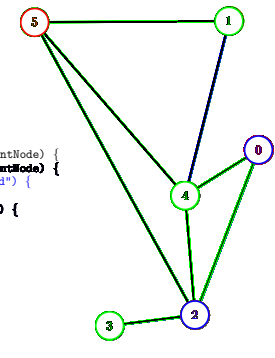
4

Breadth First Search

```

bfs(graph, node) {
  List state(graph.nodes(), "undiscovered")
  List parent(graph.nodes())
  state[node] ← "discovered"
  parent[node] ← nil
  Queue q
  q.enqueue(node)
  while (!q.isEmpty()) {
    currentNode ← q.dequeue()
    processVertexEarly(currentNode)
    state[currentNode] ← "processed"
    for each neighbour: e in Neighbourhood(currentNode) {
      if (state[neighbour] == "undiscovered") {
        processEdge(currentNode, neighbour)
        if (state[neighbour] == "undiscovered") {
          state[neighbour] ← "discovered"
          parent[neighbour] ← currentNode
          q.enqueue(neighbour)
        }
      }
    }
    processVertexLate(currentNode)
  }
  return parent
}

```



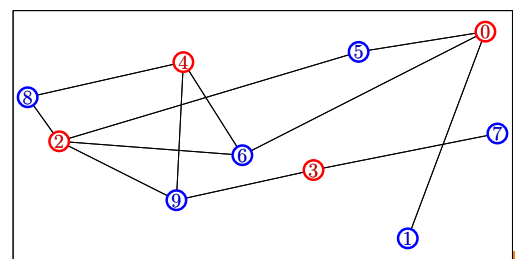
AICE1005

Algorithms and Analysis

6

Bipartite Graphs

- Bipartite graphs are graphs where the vertices can be split into two sets so that there are no edges between vertices in the same graph



AICE1005

Algorithms and Analysis

8

- Each edge must connect nodes from different sets

```
isBipartite(graph) {
  colour = List(graph.noNodes(), "uncoloured");
  bipartite = true;

  foreach node in graph {
    if (colour[node] == "uncoloured") {
      colour[node] = white;
      bfs(graph, node);
    }
  }
  return bipartite;
}

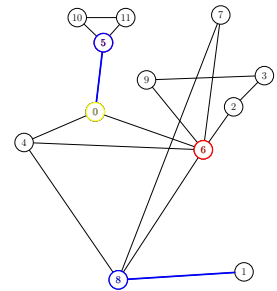
processEdge(node1, node2) {
  if (colour[node1] == colour[node2])
    bipartite = false;
  colour[node2] = (colour[node1] == "white") ? "black" : "white";
}
```

AIICE1005

Algorithms and Analysis

9

- Breadth First Search
 - BFS applications
- Depth First Search
 - DFS applications
- Topological Sort



AIICE1005

Algorithms and Analysis

10

Depth First Search

- Depth first search is essentially like breadth first search except we replace the queue by a stack
- In practice it is often implemented using recursion rather than a stack
- It proves useful to keep a record of the traversal **time** for each vertex
 - the clock ticks each time a vertex is entered or exited

AIICE1005

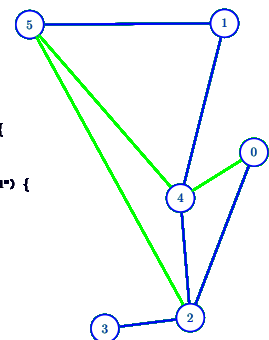
Algorithms and Analysis

11

Depth First Search

```
dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexEarly(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] == "undiscovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") {
      processEdge(node, neighbour)
    }
    if ("finished") return
  }
  processVertexLate(currentNode)
  state[currentNode] ← "processed"
  time ← time + 1
}
```

node=0 neighbour=11 time=00



AIICE1005

Algorithms and Analysis

12

Applications of DFS

- Depth first search has many applications
- Suppose we want to check if the graph is a tree (i.e. has no cycles)
- The only edges that are allowed are parent edges

```
processEdges(node1, node2) {
  if (parent[node1] ≠ node2) {
    isTree ← false
    finish ← true
  }
}
```

(note that we set `finish` to stop DFS prematurely)

AIICE1005

Algorithms and Analysis

13

Biconnected Graphs and Articulation Vertices

- If we removed a vertex (and all its edges) from a graph would the graph become disconnected?
- Such nodes are called **articulation vertices**
- Graphs with no articulation vertices are said to be **biconnected**
- In many applications articulation vertices are important, e.g. they represent single point of failures in communication networks
- A brute force method for identifying articulation vertices is to remove each and check for connectivity—this would take $O(n(m + n))$ time. Can we do this any faster?

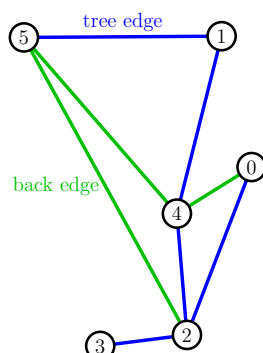
AIICE1005

Algorithms and Analysis

14

Single Pass Algorithm

- In DFS we divide the edges into tree edges that define the search tree (edges between nodes and parents) and back edges which take us back to vertices we have already seen
- Without the back edges we have a tree where all non-leaf nodes are articulation nodes
- The back edges secure the edges to the rest of the tree



AIICE1005

Algorithms and Analysis

15

Reachable Ancestors

- Need to check if there exist back edges to nodes that have been visited earlier
- We maintain an array noting the reachable ancestors of all nodes
- This is initialised in the `processVertexEarly` method to the node itself
- In the `processEdge` method, if the edge is a **back edge** we update the reachable ancestor (we check the `entryTime` to see if the edge leads to a vertex which was discovered earlier)
- tree edge** we maintain a count of the number of tree edges connected to the vertex (used to determine if we are at a leaf node)

AIICE1005

Algorithms and Analysis

16

Types of Articulated Vertices

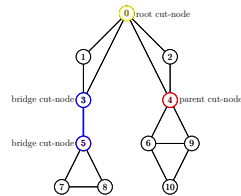
- Key is to recognise that articulated vertices only occur in three version

Root cut-nodes Occur when the root has more than one child

Bridge cut-nodes Occurs when the earliest reachable vertex (not including the tree edge to the parent) is the vertex itself. The parent will be an articulation node as will be the node itself if it is not a leaf node

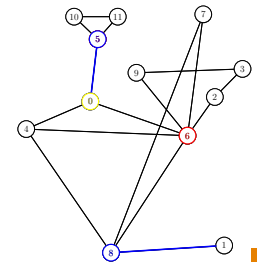
Parent cut-nodes If the earliest reachable vertex is its parent then the parent is an articulation node

- These are determined in `processVertexLate` method.



Biconnectivity Summary

- Algorithmic details are not too important
- One pass ($O(n + m)$) algorithm
- Uses `processVertexEarly`, `processEdge` and `processVertexLate` methods
- Bridge cuts also shows articulation edges



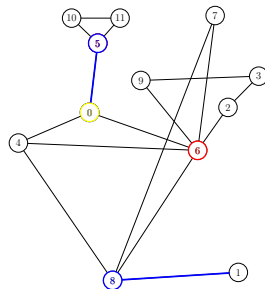
AIICE1005

Algorithms and Analysis

17

Outline

- Breadth First Search
 - BFS applications
- Depth First Search
 - DFS applications
- Topological Sort



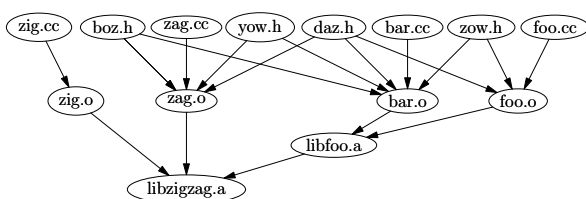
AIICE1005

Algorithms and Analysis

19

Program Compilation

- One example of a DAG is in compiling programs
- Some programs depend on other programs so they need compiling first



AIICE1005

Algorithms and Analysis

21

Topological Sort

- Given a DAG a topological sort outputs an ordered list of vertices which respects the ordering imposed by the edges
- That is, for each edge (i, j) , vertex i will occur before vertex j
- Any DAG will have at least one topological sort, but most DAGs will have many topological sorts
- Topological sort is not a "sort", but it is a useful algorithm for some applications

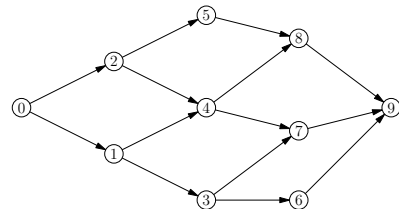
AIICE1005

Algorithms and Analysis

23

DAGs

- Directed acyclic graphs or DAGs are directed graphs without cycles
- They are often used to represent complex processes
 - Vertices are processes
 - Directed edge (i, j) indicates process i needs to occur before process j



AIICE1005

Algorithms and Analysis

20

Other Applications

- The same problem occurs in compiling classes
 - The implementation of a class can depend on the implementation of other graphs
 - What order should you compile the classes?
- In taking a degree various modules have other modules as prerequisites—what order should a student study the modules in?
- In your graduation ceremony there is the VC, Provost, Dean, HOS, Professors, etc.—in what order should they process?
- If the graphs were not acyclic it would be impossible process them!

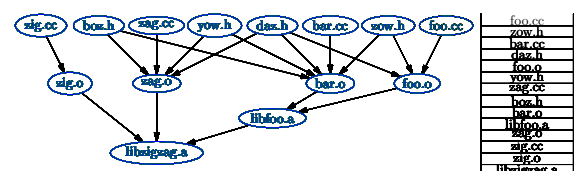
AIICE1005

Algorithms and Analysis

22

Performing a Topological Sort

- A topological sort is generated by a reversed order list of how DFS processes nodes



AIICE1005

Algorithms and Analysis

24

```

DFS on directed graphs

dfs(graph, node) {
  if ("finished") return
  state[node] ← "discovered"
  time ← time + 1
  processVertexFirst(node)
  foreach neighbour ∈ Neighbourhood(node) {
    if (state[neighbour] ≠ "discovered") {
      parent[neighbour] ← node
      processEdge(node, neighbour)
      dfs(graph, neighbour)
    } else if (state[neighbour] ≠ "processed") { (graph is directed) }
    processEdge(node, neighbour)
  }
  if ("finished") return
}
processVertexLast(currentNode)
state[currentNode] ← "processed"
time ← time + 1
}

```

Enhance DFS

- Requires us to define a couple of helper function

```

processVertexLast(node) {
  stack.push(node)
}

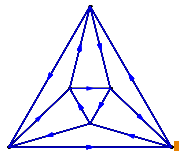
processEdge(currentNode, neighbour) {
  if (state[neighbour] == "processed") {
    print "error:_graph_not_a_DAG"
    finished = true
  }
}

```

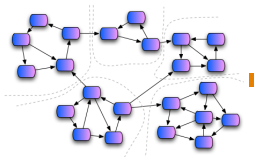
Other Applications

- DFS is used for many other classic problems

- Euler Cycles



- Strongly Connected Components



- Given our DFS programme we define

```

topologicalSort(graph) {
  Stack stack
  for node ∈ graph.vertexSet()
    if (¬discovered[node])
      dfs(graph, node)

  List topSortList
  while (¬stack.isEmpty())
    topSortList.add(stack.pop())

  return topSortList
}

```

Implementation Issues

- Most awkward part of the implementation is that the topologicalSort algorithm needs access to dfs structures (discovered[])

- processVertexLast(node) needs access to the stack

- Need to be able to redefine processVertexFirst, processEdge and processVertexLast

- Different languages and libraries cope with this differently

- ★ Java: JDSDL, JGraphT
- ★ C++: Boost Graph Library, LEDA

Lessons

- Breadth first and depth first search are different methods for traversing graphs
- They are used as part of many specific algorithms for discovering graph properties
- Breadth first search is particularly important for finding shortest paths in unweighted graphs
- Depth first search is used in a whole host of applications (finding articulation points, Euler cycles, strongly connected components)
- One of the most used application is in topological sort (finding an ordering of processes represented by a DAG)