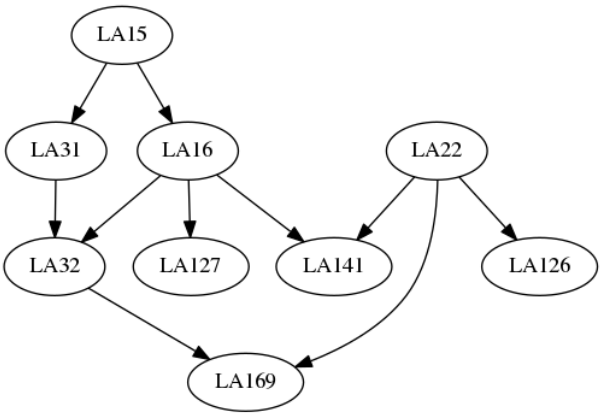
Data Structured and Algorithms
Msc Program 2017-2018

Giannis Tsagatakis
4th Assignment: Graphs

24 Μαΐου 2018

R-6.4

Bob loves foreign languages and wants to plan his course schedule to take the following nine language courses: LA15, LA16, LA22, LA32, A126, LA127, LA141, LA169. Given the course prerequisities, find a sequence of courses that allows Bob to satisfy all the prerequisities.



Σχήμα 1: Course prerequisites

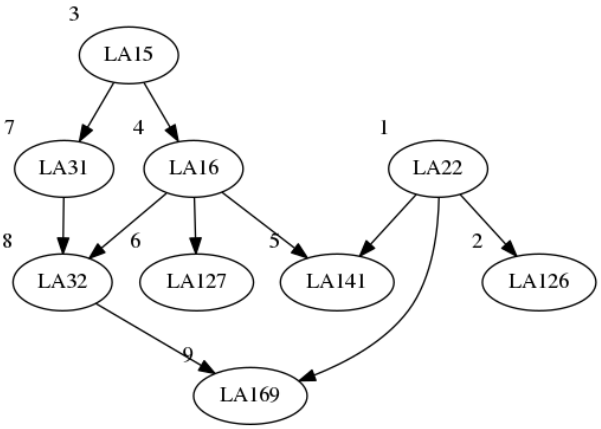
Solution

Solution using the topological sorting algorithm.

- First we build a digraph to represent the course prerequisite requirements.
- Then apply the topological sorting algorithm on the graph.

Implementation

```
#include <fstream>
#include <iostream>
#include <sstream>
```



Σχήμα 2: Course prerequisites solution.

```
#include <list>
#include <stack>
#include <vector>

#include <algorithm>
#include <iterator>

using nodeIndex = unsigned int;
using std::stack;

class Graph;
void printStack(Graph &graph, stack<nodeIndex> &Stack);

// Class to represent a graph
class Graph {
public:
  enum class State { UNVISITED, VISITED };

  // The Constructor
  explicit Graph(nodeIndex V) {
    this->V = V;

    adj = new std::list<nodeIndex>[V];
    data.resize(V);
    states.resize(V);
    order.resize(V);
    for (auto i = 0; i < V; i++) {
      states[i] = State::UNVISITED;
      data[i] = std::string("n") + std::to_string(i);
      order[i] = -999;
    }
  }

  // Name a node
  void setData(nodeIndex v, std::string name) {
```

```
  data[v] = std::move(name);
}

// Get node Data
std::string getNode(nodeIndex v) {
  return data[v];
}

// Connect 2 Nodes
void addEdge(nodeIndex v, nodeIndex w) {
  adj[v].push_back(w);
}

// Make a png file
// This needs graphviz installed
void saveAsPNG(const std::string &fname, bool showOrder = false) {
  std::stringstream ss;
  ss << "digraph {\n";
  for (int i = 0; i < V; i++) {
    if (showOrder)
      ss << "  " << i << " [label=\"" << data[i] << "\", xlabel=\""
         << order[i] << "\"]\n";
    else
      ss << "  " << i << " [label=\"" << data[i] << "\"]\n";
    for (auto j : adj[i]) {
      ss << "  " << i << " -> " << j << "\n";
    }
  }
  ss << "}";
  std::ofstream os("graph1.dot");
  os << ss.str();
  os.close();

  auto cmd = std::string("dot -Tpng graph1.dot -o ") + fname;
  std::system(cmd.c_str());
}

// The Driver function
std::stack<nodeIndex> topo_Sort() {
  stack<nodeIndex> visitStack;
  // Mark all nodes as unvisited
  std::fill(states.begin(), states.end(), State::UNVISITED);

  for (nodeIndex i = 0; i < V; i++) {
    if (states[i] == State::UNVISITED) {
      __topo__Sort(i, visitStack);
    }
  }

  // Number the nodes for graphviz visualization
  auto tmpStack = visitStack;
```

```
    auto i = 1;
    while (!tmpStack.empty()) {
      auto it = tmpStack.top();
      tmpStack.pop();
      order[it] = i++;
    }

    return visitStack;
  }

private:
  nodeIndex V;
  std::list<nodeIndex> *adj;
  std::vector<std::string> data;
  std::vector<State> states;
  std::vector<int> order;

private:
  void __topo__Sort(nodeIndex v, stack<nodeIndex> &visitStack) {
    states[v] = State::VISITED;

    for (auto node : adj[v]) {
      if (states[node] == State::UNVISITED) {
        __topo__Sort(node, visitStack);
      }
    }
    visitStack.push(v);
  }
}; // End class Graph

void printStack(Graph &graph, stack<nodeIndex> &Stack) {
  auto copy = Stack;
  bool first = true;
  while (!copy.empty()) {
    if (!first)
      std::cout << " -> ";
    std::cout << graph.getNode(copy.top());
    copy.pop();
    first = false;
  }
  std::cout << "\n";
}

// Driver program to test above functions
int main() {
  // Create a graph given in the above diagram
  Graph courses(9);
  courses.setData(0, "LA15");
  courses.setData(1, "LA31");
  courses.setData(2, "LA16");
  courses.setData(3, "LA32");
```

```
  courses.setData(4, "LA127");
  courses.setData(5, "LA141");
  courses.setData(6, "LA22");
  courses.setData(7, "LA169");
  courses.setData(8, "LA126");

  courses.addEdge(0, 1);
  courses.addEdge(0, 2);
  courses.addEdge(1, 3);
  courses.addEdge(2, 3);
  courses.addEdge(2, 4);
  courses.addEdge(2, 5);
  courses.addEdge(6, 5);
  courses.addEdge(6, 8);
  courses.addEdge(6, 7);
  courses.addEdge(3, 7);

  courses.saveAsPNG("../gcourses.png", false);

  std::cout << "\n\nTopological Sort:\n";
  auto results = courses.topo_Sort();
  courses.saveAsPNG("../gcourses-solution.png", true);
  printStack(courses, results);

  return 0;
}
```

Coding solution

```
Topological Sort:
LA22 -> LA126 -> LA15 -> LA16 -> LA141 -> LA127 -> LA31 -> LA32 -> LA169
```

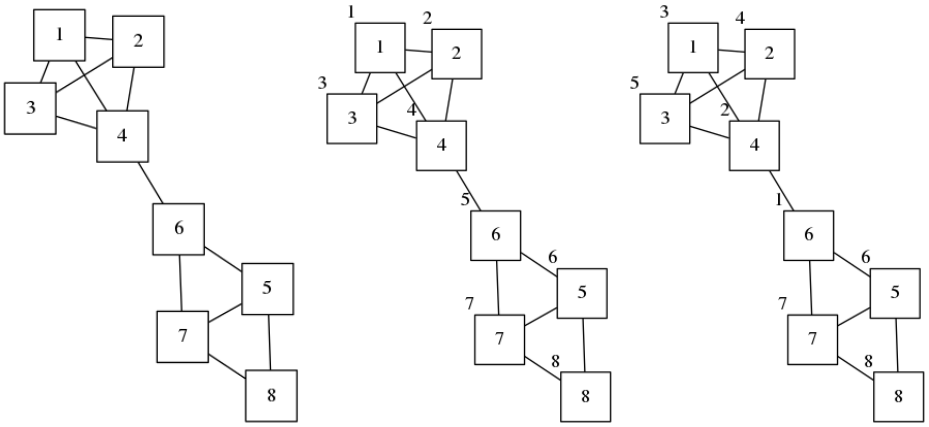
R-6.6

Let G be a graph whose vertices are the integers 1 through 8, and let the adjacent vertices of each vertex be given by the table given as code below:

```
  Graph gg(8);
  gg.addEdge(2, 0);
  gg.addEdge(2, 1);
  gg.addEdge(2, 3);
  gg.addEdge(0, 1);
  gg.addEdge(0, 2);
  gg.addEdge(0, 3);
  gg.addEdge(1, 0);
  gg.addEdge(1, 2);
  gg.addEdge(1, 3);
  gg.addEdge(3, 0);
  gg.addEdge(3, 1);
  gg.addEdge(3, 2);
  gg.addEdge(3, 5);
  gg.addEdge(4, 5);
  gg.addEdge(4, 6);
  gg.addEdge(4, 7);
  gg.addEdge(5, 3);
  gg.addEdge(5, 4);
  gg.addEdge(5, 6);
  gg.addEdge(6, 4);
  gg.addEdge(6, 5);
  gg.addEdge(6, 7);
  gg.addEdge(7, 4);
  gg.addEdge(7, 6);
```

Assume that, in a traversal of G, the adjacent vertices of a given vertex are returned in the same order as they are listed in the above table.

- Draw G.



Σχήμα 3: 2 Solutions and the original problem.

- Order the vertices as they are visited in a DFS traversal starting at vertex 1.
- Order the vertices as they are visited in a BFS traversal starting at vertex 1.

DFS Implementation

The solution share alot of code with the previous algorithm. The code is a bit more simple actually. A list is used to hold the visited nodes instead of a stack. Thus only the most important parts of the alogorithm is given bellow.

```
// The Driver function
std::list<nodeIndex> DFS_Sort(nodeIndex v = 0) {
  std::list<nodeIndex> visitList;
  std::fill(states.begin(), states.end(), State::UNVISITED);

  __dfs__Sort(v, visitList);

  // Number the nodes for graphviz visualization
  auto i = 1;
  for (auto it : visitList)
    order[it] = i++;

  return visitList;
}

void __dfs__Sort(nodeIndex v, std::list<nodeIndex> &visitList) {
  states[v] = State::VISITED;
  visitList.emplace_back(v);

  for (auto node : adj[v]) {
    if (states[node] == State::UNVISITED) {
      __dfs__Sort(node, visitList);
    }
  }
}
```

```
void printList(const std::list<nodeIndex> &results) {
  bool first = true;
  for (auto &it : results) {
    if (!first) {
      std::cout << "->";
    }
    auto name = (int)it + 1;
    std::cout << name;
    first = false;
  }
}
```

Test run

```
  auto pos = 0;

  auto results = gg.DFS_Sort(pos);
  gg.saveAsPNG("../DFS-solution.png", true);

  std::cout << "\n\nDFS Sort starting from :" << pos + 1 << "\n";
  printList(results);
```

DFS Coding solution

```
DFS Sort starting from :1
1->2->3->4->6->5->7->8

DFS Sort starting from :6
6->4->1->2->3->5->7->8)
```

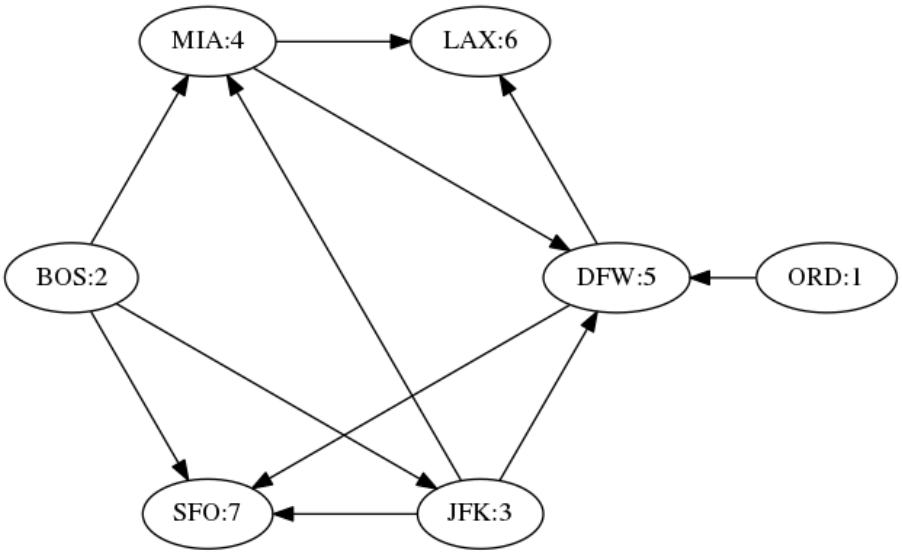
R-6.10

Compute a topological ordering for the directed graph drawn with solid edges in Figure.

Flight Implementation

The code is the almost the same as the 1st exersize.

```
  void addEdge(nodeIndex v, nodeIndex w) {
    adj[v].push_back(w);
  }

  // Make a png file
  // This needs graphviz installed
  void saveAsPNG(const std::string &fname, bool showOrder = false) {
    std::stringstream ss;
    ss << "digraph {\n";
    for (int i = 0; i < V; i++) {
      std::string label;
      if (showOrder)
        label = std::string(data[i]) + ":" + std::to_string(order[i]);
```



Σχήμα 4: Topological Sort of the flights problem.

```
      else
        label = std::string(data[i]);
      ss << "  " << i << " [label=\"" << label << "\"]\n";

      for (auto j : adj[i]) {
        ss << "  " << i << " -> " << j << "\n";
      }
    }
    ss << "}";
    std::ofstream os("graph1.dot");
```

The driver program.

```
    std::cout << graph.getNode(copy.top());
    copy.pop();
    first = false;
  }
  std::cout << "\n";
}

// Driver program to test above functions
int main() {
  // Create a graph given in the above diagram
  Graph courses(7);
  courses.setData(0, "SFO");
  courses.setData(1, "BOS");
  courses.setData(2, "ORD");
  courses.setData(3, "JFK");
  courses.setData(4, "DFW");
  courses.setData(5, "LAX");
  courses.setData(6, "MIA");

  courses.addEdge(1, 3);
  courses.addEdge(1, 0);
```

```
courses.addEdge(1, 6);
courses.addEdge(2, 4);
courses.addEdge(3, 0);
courses.addEdge(3, 4);
courses.addEdge(3, 6);
courses.addEdge(4, 0);
courses.addEdge(4, 5);
courses.addEdge(6, 4);
courses.addEdge(6, 5);

courses.saveAsPNG("../gflights.png", false);
```

Flight Coding solution

```
Topological Sort:
ORD -> BOS -> JFK -> MIA -> DFW -> LAX -> SFO
```

R-6-7

Would you use the adjacency list structure or the adjacency matrix structure in each of the following cases? Justify your choice.

Solution

1. The graph has 10,000 vertices and 20,000 edges, it is important to use as little space as possible

   The adjacency matrix needs space of $10,000 \times 10,000 = 100,000$ elements for a directed graph and the half for an undirected graph. The adjacency list structure needs only storage for $20,000$ edge elements (the data and the pointers). Clearly the adjacency list structure is preferable in this case.

2. The graph has 10,000 vertices and 20,000,000 edges, and it is important to use as little space as possible.

   There is no clear winner here. The exact space requirements depends on immplementation details like the size of a pointer. In this case it will be better if we use criteria like the next question to decide.

3. You need to answer the query `areAdjecent` as fast as possible, no matter how much space you use.

   The adjacency matrix structure is much better for operation `areAdjacent` , while the adjacency list structure is much better for operations `insertVertex` and `removeVertex` . Clearly here the adjacency matrix structure is preferable. Indeed, it supports operation `areAdjacent` in $O(1)$ time, irrespectively of the number of vertices or edges.

R-6-9

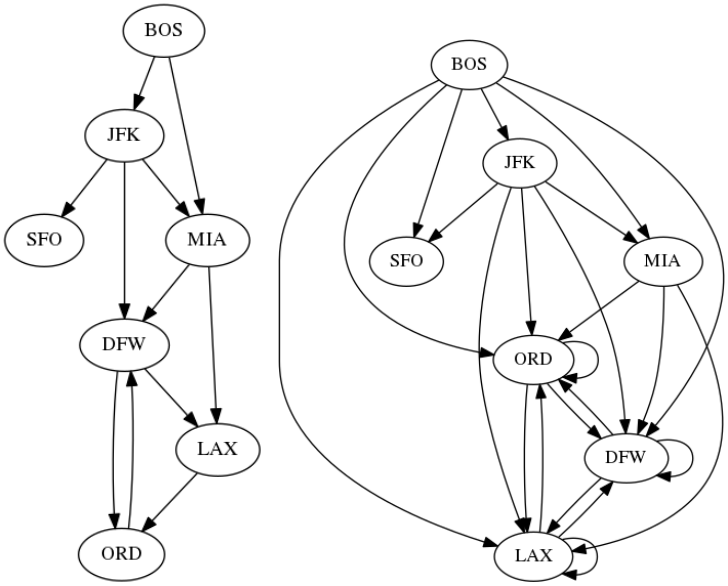
Draw the transitive closure of the directed graph shown in Figure

Transitive closure Implementation

```
#include <fstream>
#include <iostream>
#include <sstream>

#include <list>
#include <stack>
#include <vector>

#include <algorithm>
#include <iterator>

using nodeIndex = unsigned int;

// Class to represent a graph
class Graph {
public:
  explicit Graph(nodeIndex V) {
    this->V = V;
    adj = new std::list<nodeIndex>[V];
    data.resize(V);
    for (auto i = 0; i < V; i++) {
      data[i] = std::string("n") + std::to_string(i);
    }
  }

  void setData(nodeIndex v, std::string name) {
    data[v] = std::move(name);
  }

  std::string getNode(nodeIndex v) {
    return data[v];
  }

  void addEdge(nodeIndex v, nodeIndex w) {
    adj[v].push_back(w);
  }

  void saveAsPNG(const std::string &fname) {
    std::stringstream ss;
    ss << "digraph {\n";
    for (int i = 0; i < V; i++) {
      ss << "  " << i << " [label=\"" << data[i] << "\"]\n";
      for (auto j : adj[i]) {
        ss << "  " << i << " -> " << j << "\n";
      }
    }
    ss << "}";
    std::ofstream os("trans.dot");
    os << ss.str();
```

```
    os.close();

    auto cmd = std::string("dot -Tpng trans.dot -o ") + fname;
    std::system(cmd.c_str());
  }

  int isConnected(nodeIndex i, nodeIndex j) {
    auto connections = adj[i];
    bool found = (std::find(connections.begin(), connections.end(), j) !=
                  connections.end());
    if (found)
      return 1;
    return 0;
  }

  void transitiveCloure() {
    // reach[][] will be the DP matrix, each element
    // will have the shortest distance between pairs.
    int reach[V][V];

    for (nodeIndex i = 0; i < V; i++)
      for (nodeIndex j = 0; j < V; j++)
        reach[i][j] = isConnected(i, j);

    // For algorithm look at page 636 of Cormen (greek version)
    for (auto k = 0; k < V; k++)
      // Pick all vertices as source
      for (auto i = 0; i < V; i++)
        // Pick all vertices as destination
        for (auto j = 0; j < V; j++) {
//          reach[i][j] =
//              isConnected(i, j) || (isConnected(i, k) && isConnected(k, j));
          reach[i][j] = reach[i][j] || (reach[i][k] && reach[k][j]);
        }

    // Print table in latex format
    std::cout << "\t&";
    for (nodeIndex j = 0; j < V; j++)
      std::cout << data[j] << "\t&";
    std::cout << "\\\\\n";
    std::cout << "\\hline\\hline\n";

    for (nodeIndex i = 0; i < V; i++) {
      std::cout << data[i] << "\t&";
      for (nodeIndex j = 0; j < V; j++)
        std::cout << reach[i][j] << "\t\t&";
      std::cout << "\\\\\n";
    }


    // TODO: Create a copy constructor.
```

```
    // Create a new graph
    Graph gtrans(V);
    for(auto l=0; l<V; l++){
      gtrans.setData(l,getNode(l));
    }

    for (auto i = 0; i < V; i++)
      for (auto j = 0; j < V; j++)
        if (reach[i][j] )
          gtrans.addEdge(i,j);

    // Print the transitive graph
    gtrans.saveAsPNG("../gflight-trans.png");
  }

private:
  nodeIndex V;
  std::list<nodeIndex> *adj;
  std::vector<std::string> data;
}; // End class Graph

// Driver program to test above functions
int main() {
  Graph g(4);

  Graph flights(7);
  flights.setData(0, "SFO");
  flights.setData(1, "BOS");
  flights.setData(2, "ORD");
  flights.setData(3, "JFK");
  flights.setData(4, "DFW");
  flights.setData(5, "LAX");
  flights.setData(6, "MIA");

  flights.addEdge(1, 3);
  flights.addEdge(1, 6);
  flights.addEdge(2, 4);
  flights.addEdge(3, 0);
  flights.addEdge(3, 4);
  flights.addEdge(3, 6);
  flights.addEdge(4, 2);
  flights.addEdge(4, 5);
  flights.addEdge(5, 2);
  flights.addEdge(6, 4);
  flights.addEdge(6, 5);

  flights.saveAsPNG("../gflight.png");
  flights.transitiveCloure();

  return 0;
}
```



Σχήμα 5: The transitive close of the flights problem.

	SFO	BOS	ORD	JFK	DFW	LAX	MIA
SFO	0	0	0	0	0	0	0
BOS	1	0	1	1	1	1	1
ORD	0	0	1	0	1	1	0
JFK	1	0	1	0	1	1	1
DFW	0	0	1	0	1	1	0
LAX	0	0	1	0	1	1	0
MIA	0	0	1	0	1	1	0

Tools used: `clang++`, `clion`, `cmake`, LaTeX, `clang-format`, `graphviz dot`.