DAG-TopologicalSort

August 7, 2020

1 Topological Sort

- https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GraphTopsort.html
- the process of laying out the vertices of a directed acyclic graph (DAG) in a linear order to meet the prerequisite rules

1.1 Applications of Topological Sort

- scheduling a series of tasks, such as classes or construction jobs, where certain task cannot be started until after prerequisities are completed
- wish to organize the tasks in a linear order that allows us to complete them one at a time without violating any prerequisities
- model the problem using a DAG
  - graph is directed because one task is a prerequisite of another
  - graph is acyclic becasue a cycle would indicate a conflicting series of prereq that couldn't be completed wihout violating at least one prereq
- e.g. in Figure 1, seven tasks have dependencies as shown by the directed graph:
  Fig. 1 DAG
- an acceptable topological sort for Figure 1 graph is: J1, J2, J3, J4, J6, J5, J7

1.2 Depth-first Solution; stack-based

- a topo sort can be found by performing a DFS on the graph
- algorithm steps:
  - when a vertex is visited, no action is taken (i.e., function PreVisit does nothing)
  - when the recursion pops back to that vertex, function PostVisit prints the vertex
  - yields a topological sort in reverse order
  - use stack to put it back in the right order
  - it doesn't not matter where the sort starts, as long as all the vertices are visited in the end
- visualize it here: https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GraphTopsort.html

[1]:
```cpp
// we'll use unordered_map to represent graph
// allows us to store node with any data type without creating/using graph ADT
#include <unordered_map>
#include <vector>
#include <iostream>
#include <string>
#include <utility>
```

```
#include <stack>
#include <queue>
using namespace std;
```

[2]:
```
// operator<< overloaded to print a vector
template<class T>
ostream& operator<<(ostream& out, const vector<T>& v) {
    char comma[3] = {'\0', ' ', '\0'};
    out << '[';
    for (auto& e: v) {
        out << comma << e;
        comma[0] = ',';
    }
    out << "]\n";
    return out;
}
```

[3]:
```
// operator<< overloaded to print a unordered_map container
template<class T1, class T2>
ostream& operator<<(ostream& out, const unordered_map<T1, T2>& m) {
    //char comma[3] = {'\0', ' ', '\0'};
    out << "{\n";
    for (auto& e: m) {
        out << "    " << e.first << ':' << e.second;
        //comma[0] = ',';
    }
    out << "}\n";
    return out;
}
```

[4]:
```
// Lets generate the above graph using unordered_map
// key is the node and and vector of value is its neighbors
// Creates out-degree based graph; adjacency-list
using Graph = unordered_map<string, vector<string> >;
Graph DAG;
```

[5]:
```
DAG.insert(make_pair("J1", vector<string>{"J2", "J3"}));
DAG.insert(make_pair("J2", vector<string>{"J6", "J4"}));
DAG.insert(make_pair("J3", vector<string>{"J4"}));
DAG.insert(make_pair("J4", vector<string>{"J5"}));
DAG.insert(make_pair("J5", vector<string>{"J7"}));
DAG.insert(make_pair("J6", vector<string>()));
DAG.insert(make_pair("J7", vector<string>()));
DAG["J2"].push_back("J5");
```

[6]:
```
// lets check properties of the graph
cout << "total nodes of DAG = " << DAG.size() << endl;
```

```
total nodes of DAG = 7
```

[7]:
```
// print the no. of neighbors of J2
cout << "total neigbors of J2 = " << DAG["J2"].size() << endl;
```

```
total neigbors of J2 = 3
```

[8]:
```
// print the whole DAG
cout << DAG;
```

```
{
   J6:[]
   J4:[J5]
   J1:[J2, J3]
   J5:[J7]
   J3:[J4]
   J7:[]
   J2:[J6, J4, J5]
}
```

[9]:
```
unordered_map<string, bool> visited;
for(auto pair: DAG) {
    visited[pair.first] = false;
}
```

[10]:
```
// answer stack stores the topological order of the nodes
void DFS(Graph& G, unordered_map<string, bool>& visited, string node,␣
 ↪stack<string>& answer) {
    // mark node as visited
    visited[node] = true;
    // run DFS on all its neighbors
    for (auto& neighbor: G[node]) {
        if (!visited[neighbor]) {
            DFS(G, visited, neighbor, answer);
        }
    }
    // visit/print node
    answer.push(node);
}
```

[11]:
```
// depth-first topological sort
void TopologicalSort(Graph& G, stack<string>& answer) {
    // visited unordered_map, <node, visited>
    unordered_map<string, bool> visited;
    // mark each node as not visited
    for(auto& pair: G) {
        visited[pair.first] = false;
    }
```

```
    // run DFS from each node if that node is not visited
    for(auto& pair: G) {
        if (!visited[pair.first]) {
            DFS(G, visited, pair.first, answer);
        }
    }
}
```

[12]:
```
stack<string> answer;
```

[13]:
```
// run topological sort or DAG
TopologicalSort(DAG, answer);
```

[14]:
```
// print the topological sort from answer stack
while(!answer.empty()) {
    cout << answer.top() << " ";
    answer.pop();
}
```

```
J1 J3 J2 J4 J5 J7 J6
```

1.3 Queue-based Solution - Kahn's Algorithm

- in-degree-based solution
- https://en.wikipedia.org/wiki/Topological_sorting
- See https://opendsa-server.cs.vt.edu/ODSA/Books/CS3/html/GraphTopsort.html

1.3.1 Algorithm Steps

1. compute in-degree (number of incoming edges/prerequisites) for each vertex
2. Place vertices with no prerequisitis or in-degrees of 0 into a queue
3. While queue is not empty:
    1. pop the next vertex, v from the queue; and print it
    - decrese in-degree of v's neighbors (i.e. all vertices that hav v as a prerequisite) by 1
    - if in-degree of the neighbor becomes 0, add it to the queue
4. If the count of printed/visited nodes is NOT equal to the number of nodes, graph contains cycle or topological sort not possible

1.4 E.g., let's work on the above DAG in Fig. 1

[15]:
```
void KahnsAlgorithm(Graph &G, vector<string> &answer) {
    // step 1: computer in-degree
    // this can be done when building graph for efficiency!
    unordered_map<string, int> indegree;
    for (auto &pair: G)
        indegree[pair.first] = 0; // initialize in-degree of each v to 0

    for (auto &pair: G) {
```

```
        for (auto neighbor: G[pair.first]) // for each neighbor of v
            indegree[neighbor] += 1;
    }

    // step 2: create a queue of all vertices with 0-indegree
    queue<string> tasksQ;
    for(auto & pair: indegree) {
        if (indegree[pair.first] == 0)
            tasksQ.push(pair.first);
    }

    // step 3:
    while(!tasksQ.empty()) {
        string v = tasksQ.front(); // access the first element
        tasksQ.pop(); // remove the first element; step 3.A
        answer.push_back(v);
        // 3.B
        for(auto n: G[v]) {
            --indegree[n];
            // 3.C
            if (indegree[n] == 0)
                tasksQ.push(n);
        }
    }
    // step 4
    //cout << "answer-size = " << answer.size() << endl;
    if (answer.size() != G.size())
    {
        // topological sort not possible
        answer.clear();
        answer.push_back("topological sort not possible!");
    }
}
```

[16]:
```
// let's test the KahnsAlgorithm
vector<string> ans;
```

[17]:
```
KahnsAlgorithm(DAG, ans);
cout << ans;
```

```
[J1, J2, J3, J6, J4, J5, J7]
```

1.5 Exercises

1. Brexit - https://open.kattis.com/problems/brexit

- Build Dependencies - https://open.kattis.com/problems/builddeps
- Running MoM - https://open.kattis.com/problems/runningmom

  - finding cycle in a DAG
- Conservation - https://open.kattis.com/problems/conservation
  - Hints: Keep two queues; one for each lab
  - Run Topoligical sort twice one starting from lab1 and another from lab2
  - use the min answer
- Managin Package - https://open.kattis.com/problems/managingpackaging
  - Hints: Create in-degree/dependency graphs
  - Use queue-based topoligical sort; replace queue with priority queue to print in sorted order

[ ]:
```

```