# Practical Session No. 12 – Graphs, BFS, DFS Topological Sort

|  |  |
| --- | --- |
| **Graphs and BFS** | |
| Graph  G = (V, E) | V(G) = V - Set of all vertices in G E(G) = E - Set of all edges (u,v) in G, where u,v in V(G) |
| Graph Representations | * Adjacency Matrix: Dimensions  M[i,j] = 1 - (vi, vj) in E(G) M[i,j] = 0 - (vi, vj) not in E(G) * Adjacency List: Each vertex v in V(G) has a list of all vertices u in V(G) where (v,u) in E(G) |
| BFS(G, s) | Breadth-First Search  Starting from the source node *s*, BFS computes the minimal distance from *s* to any other node *v* that can be reached from s. The algorithm builds a *breadth-tree* rooted at *s* with the minimal paths to nodes that can be reached from s. The algorithm is suitable for directed or undirected graphs.   * color[v] - the color of node *v*   + **white** - an unvisited node. The first time it is encountered during the search it becomes nonwhite.   + **gray** - a visited node which has white neighbors (draws a frontier between the visited and unvisited nodes)   + **black** - a node whose neighbors were visited (colored gray or black) * pred[v] - the predecessor of *v* in the breadth-tree * d[v] - the minimal distance from s to v  Time complexity: O(|V| + |E|) |

|  |  |
| --- | --- |
| **DFS, Topological-Sort** | |
| DFS(G) | Depth-First Search   * Strategy: search "deeper" in the graph whenever possible. * Edges are explored out of the most recently discovered vertex *v* that still has unexplored edges leaving it. When all of *v*'s edges have been explored, the search "backtracks" to explore edges leaving the vertex from which *v* was discovered. This process continues until we have discovered all the vertices that are reachable from the original source vertex. * If any undiscovered vertices remain, then one of them is selected as a new source and the search is repeated from that source. This entire process is repeated until all vertices are discovered. * The DFS may result in several trees, depending on the order of choosing the first vertex in the set of unvisited vertices. * color[v] - the color of node v   + white - an unvisited node   + gray - a visited node   + black - a node whose adjacency list has been examined completely (whose descendants were all searched). * pred[v] - the predecessor of v in the one of the depth trees * d[v] - the time when v was discovered * f[v] - the time when v was colored in black   A node *v* is white before d[v], gray between d[v] and f[v] and black after f[v].  Time complexity: O(|V| + |E|) |
| Topological-Sort | Ordering of vertices in a directed acyclic graph (DAG) G=(V,E) such that if there is a path from v to u in G, then v appears before u in the ordering. There could be many solutions, for example:   1. call DFS to compute f[v] 2. As the visit in each vertex is finished (blackened), insert it to the head of a linked list 3. Return the linked list of the vertices  Time Complexity: O(|V|+|E|) |

|  |  |
| --- | --- |
| DFS Edges Classifications | DFS can be used to classify the edges of the input graph G=(V,E).   1. Tree-edges (u,v) - if v was initially discovered using edge from u to v 2. Back-edges (u,v) - if v is an ancestor of u in the depth-tree 3. Forward-edges (u,v) - not a tree-edge, where u is v's ancestor 4. Cross-edges (u,v) - All the other edges, where u and v are vertices in different depth-tree, u is not v's ancestor or v is not u's ancestor   It can be shown that if the graph is undirected then all of its edges are tree edges or back edges. Therefore we define Forward-edges and Cross-edges only in directed graphs |

### Question 1

Given an undirected graph G = (V,E), (G is connected). For every 2 sets of vertices V1 and V2 such as: we define:

***distance(u,v)* -** the length of the shortest path from *u* to *v* in G.

***distance(*V1, V2*)* -** the length of shortest path between a vertex *v1* in V1 and *v2* in V2.

  (**Note**: If V1 ∩ V2 ≠  then distance(V1, V2) = 0).

Find distance(V1, V2) in O(|V| + |E|) time.

#### Solution:

We define a new Graph G'=(V',E') from G = (V,E).

We add 2 new vertices *s* and *t* to V, connect *s* to every vertex in V1, connect every vertex in V2 to *t* and the run BFS to find the length of the shortest path from *s* to *t*. The answer we get is *distance(*V1, V2*)* + 2

1. V' 🡨 V {*s,t*}
2. E' 🡨 E {(*s,u*) : }  {(*v,t*) :  }
3. Execute BFS from *s* // Finding d(*t*).
4. return d(*t*) − 2

### Question 2

Update the BFS algorithm so that every vertex *v* in the graph contains not only d[v], the length of shortest path from s to v, but also the number of different paths of that length.

#### Solution:

Each node *u* in the graph will contain an extra field:

*M(u)* - the number of different paths to u of length d(u).

1. Initialize M(v)🡨0 for all the vertices except *s*. initialize M(s)🡨1.
2. For each vertex *u,* when going over the neighbors *v* of *u*:
3. if *v* is a white neighbor then  
    M(v) 🡨 M(u)
4. else if d(v) = d(u)+1 // if v is not a white neighbor of u  
    M(v) 🡨 M(v) + M(u)

### Question 3

Given an undirected graph G=(V,E) with a weight function on the edges.

1. Suggest an O(V+E) time algorithm to find the weight of the lightest path (i.e. the path for which the sum of edge weights is minimal) from a given vertex s in V to another vertex t in V.
2. How can you adjust you algorithm so that it will also print that path?

#### Solution:

**a.** We use reduction. We turn G=(V,E) into an unweighted undirected graph

such that we can use BFS to solve the problem.

Building from G

* Init to E and to V
* For each edge (u,v) of weight 2 in E
  + Add a new vertex uv to
  + Remove (u,v) from
  + add the edges (u,uv) and (uv,v) to

2

1

1

2

1

Clearly if there is a path of weight w in G then there is a path of length w in (we simply have to replace moving over every edge (u,v) of weight 2 in G for moving over the edges (u,uv) and (uv,u) in .

In the same way, if there is a path of length w in between 2 vertices  then there is a path of weight w in G between those vertices.

This means running BFS from s on the graph  will give us the length of the weight of all the lightest paths in G that begin with s.

Runtime complexity:

The runtime complexity is .

The algorithm runs in

**b.**  In order to print the path in G when running BFS on  we need to be able to differentiate between nodes in that are from V and other nodes added to  in the for loop. We do this by adding a new field “original” to each vertex in, and correct the building algorithm accordingly.

Building from G

* Init to E and to V
* For each
  + Set v.original to 1
* For each edge (u,v) of weight 2 in E
  + Add a new vertex uv to
  + Set v.original to 0
  + Remove (u,v) from
  + add the edges (u,uv) and (uv,v) to

Now we can use the BFS on  to print the path (while printing only the vertices that belong to V).

PrintShortestPath(,s,t)

* if s = t then print s and return
* PrintShortestPath(,s,pred[t])
* if t.original = 1 then print t
* return

Runtime complexity:

The runtime complexity is the length of the path in . Note that the length of any simple path p in G is at most |V|. The length of the corresponding path in is at most double. Therefore the printing algorithm runs in time:

### 

### Question 4

Suggest an algorithm that determines whether a given **undirected** graph contains a cycle, that runs in time O(|V|) (note that runtime is independent of E).

#### Solution:

***Lemma***: An undirected graph is acyclic if and only if a DFS yields no back edges.

***Back edge*** (u,v) is such that *v* has already been discovered and is u's ancestor.  
(u,v) - if *v* is an ancestor of u in the depth-tree and (v,u) is not a tree-edge.

tree edges

back edge

The DFS algorithm can be modified to classify edges as it encounters them. In a depth-first search of an undirected graph *G*, every edge of *G* is either a tree edge or a back edge. if we arrive to a grey vertex (which means we arrived there through back edge) we return to a vertex we already visited and therefore the graph is cyclic.

**Acyclic(G=(V,E))**

execute DFS on the graph while:

1. counting the number of visited nodes
2. If DFS visits a grey node return "cycle"

### The algorithm runs in O(|V|) time, because in the worst case, the algorithm will explore |V|−1 before reaching a grey node (back edge) or terminating (because an undirected graph with |V| or more edges always contains a cycle).

### Question 5

2 lists are given:  
**A** - courses list. Every student must study all the courses in A.

**B** – Prerequisites. B contains tuples (a, b) (where *a* and *b* are in A), indicating that course *a* must be taken before course *b*. The prerequisites dependencies are acyclic.

Design a schedule for each of the following students.

1. A lazy student who wants to take only one course in a semester.
2. A student who wants to take all the courses in **A** in the minimal number of semesters. The student is willing to take any number of courses in a semester.

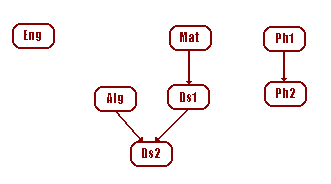
##### Example:

A = { Alg, Eng, Ds1, Ds2, Mat, Ph1, Ph2 }  
B = { (Alg, Ds2), (Ds1, Ds2), (Mat, Ds1), (Ph1, Ph2) }

|  |  |  |
| --- | --- | --- |
| Optional output for student no. 1: Semester 1: Eng Semester 2: Mat Semester 3: Alg Semester 4: Ds1 Semester 5: Ds2 Semester 6: Ph1 Semester 7: Ph2 |  | Optional output for student no. 2: Semester 1: Eng, Mat, Ph1, Alg Semester 2: Ds1, Ph2 Semester 3: Ds2 |

#### Solution:

Assume that the courses names are integers in the range [1..n], n is known (n is not constant). The relations between the courses will be represented by a **directed** graph G = (V, E), where V is the set of courses and if course *i* is a prerequisite of course *j*, E will contain the edge *(i, j)*. The graph will be represented as an adjacency list.

The graph for the given example is:  


**Algorithm for a lazy student:** Print the courses in a way that course i won't be printed before course j, if j is a prerequisite of i. In other words, sort the courses topologically.  
Complexity: O(|V| + |E|)

Now, let's observe ***another*** algorithm for topological sort of a DAG in O(|V|+|E|).

* Find in-degree of all the vertices and keep a list of vertices with in-degree=0 - O(|V|+|E|)
* While V is not empty do:
  + Extract a vertex *v* with in-degree of 0 from the list - O(1)
  + output *v* and remove it from G, along with its edges,
  + Reduce the in-degree of each node *u* such as (*v,u*) was an edge in G and add to the list vertices with in-degree=0, if necessary - O(d(v))

**Algorithm for a hard working student:** A variation of the above topological sort algorithm with a slight change. In each semester k, execute the algorithm on all the nodes with degree 0 simultaneously (instead of dealing with one source at each stage, all the sources will be dealt and printed in one semester.)  
(note that the topological sort is not necessary because after we find the vertexes with in-degree 0, we can go to the next vertexes through the edges.)

Complexity: O(|V| + |E|)

### Question 6

Given a DAG (directed acyclic graph) G=(V,E) with weighted edges. Suggest an O(|V|+|E|) time algorithm for finding the weight of the maximum weight path in G.

#### Solution:

1. Topologically sort V - *O*(V+E).

(We'll denote the topologically ordered vertex list as (*v1,v2, …, vn*)).

1. For every vertex v, create a list of all in-edges, i.e., list of all the vertices *u*, such that (*u,v*)∈E - *O*(V+E).
2. Define a new array A for which A[i] = the maximal weighted path that ends in vi. By definition, A[i] = max{A[j]+w(j,i) : (j,i) ∈E }. - *O*(V+E).
3. Start by computing A[i] for all i = 1..n, where v1, v2, …, vn is the topological ordering.
4. Since the graph was topologically sorted, when computing A[i], all the values A[j] of
5. it’s neighbors j were already computed (all i's neighbors appear before it in the topological ordering).
6. After computing A[i] for all the nodes, find vk for which A[k] is maximal.

MaxPath(G)

v1, v2, …, vn ← TopologicalSort(G)

Create-in-edges-lists()

A ← new array [1..n]

A[1] ← 0

for i ← 2 to n do

A[i] ← max{A[j]+w(j,i) : j ∈ in\_edges\_list[i] }

return max{A[1],A[2],…,A[n]}

We design the algorithm

1. Express the solution to a problem in terms of solutions to smaller problems.

2. Solve all the smallest problems first and put their solutions in a table

3. Solve the next larger problems, and so on, up to the problem, we originally wanted to

solve.

4. Each problem should be easily solvable by looking up and combining solutions of

smaller problems in the table.

Time Complexity: **O(|V|+|E|)**

**Question 7**

Definition: A ***super-sink*** is a vertex with in-degree |V|-1 and out-degree 0 (at most one **super-sink** can exist in a graph).

When an adjacency-matrix representation is used, most graph algorithms require time . Show that determining whether a *directed* graph, represented in an adjacency-matrix contains a ***super-sink*** can be done in time O(|V|).



**Solution:**

A vertex i is a super-sink if and only if M[i,j] = 0 for all j and M[j,i] = 1 for all j ≠i.  
For any pair of vertices i and j:

i

i

x x 1 x

x x 1 x

0 0 0 0

x x 1 x

* M[i,j] = 1 🡪 vertex i can't be a super-sink
* M[i,j] = 0 🡪 vertex j can't be a super-sink (i≠j)

|  |
| --- |
| **Sink(M,n)**   1. i=0, j=0, n=|V|. 2. while (j<n)   if j=i or M[i,j]=0 //  can’t be a super-sink  j🡨j+1  if M[i,j]=1 //  can’t be a super-sink  i🡨i+1   1. Check row i – it should be all zeros, //  is a super-sink 2. Check column i – it should be all ones, except M[i,i]. 3. if both conditions hold, then i is a super-sink. 4. otherwise, there is no super-sink. |

**Note**: In the *while* condition it is enough to check that j<n, since i<n anyway if we assume the graph is not reflective (there are no edges (i,i)), then there cannot be row of 1’s in the matrix. This means that the algorithm will always be above and to the right of the main diagonal.

**Time Complexity:** *while* loop takes at most *(n+n)* steps. Checking ith row and column takes *(n+n)* time. Total: T(n)= 4n=O(|V|).