Week 07: Graph Algorithms 1

Graph Algorithms

Problems on Graphs

What kind of problems do we want to solve on/via graphs?

- is the graph fully-connected?
- can we remove an edge and keep it fully-connected?
- is one vertex reachable starting from some other vertex?
- what is the cheapest cost path from *v* to *w*?
- which vertices are reachable from *v*? (transitive closure)
- is there a cycle that passes through all vertices? (circuit)
- is there a tree that links all vertices? (spanning tree)
- what is the minimum spanning tree?
- what is the maximal flow through a graph?
- …
- can a graph be drawn in a plane with no crossing edges? (planar graphs)
- are two graphs "equivalent"? (isomorphism)

Graph Algorithms

In this course we examine algorithms for

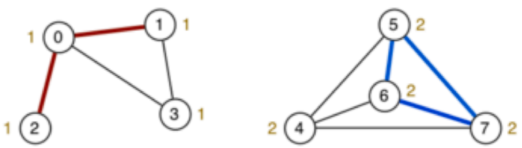
- connectivity (simple graphs)
- path finding (simple/directed graphs)
- minimum spanning trees (weighted graphs)
- shortest path (weighted graphs)
- maximum flow (weighted graphs)

We already looked at *depth-first* (DFS) and *breadth-first* (BFS) traversal …

Other DFS Examples

Other problems to solve via DFS graph search

- checking for the existence of a cycle
- determining which connected component each vertex is in



Graph with two connected components, a path and a cycle

Exercise #1: Buggy Cycle Check

A graph has a *cycle* if

- it has a path of length > 1
- with start vertex *src* = end vertex *dest*
- and without using any edge more than once

We are not required to give the path, just indicate its presence.

The following DFS cycle check has two bugs. Find them.

```
hasCycle(G):
|  Input  graph G
|  Output true if G has a cycle, false otherwise
|
|  choose any vertex v∈G
|  return dfsCycleCheck(G,v)

dfsCycleCheck(G,v):
|  mark v as visited
|  for each (v,w)∈edges(G) do
|  |  if w has been visited then   // found cycle
|  |     return true
|  |  else if dfsCycleCheck(G,w) then
|  |     return true
|  end for
|  return false                    // no cycle at v
```

1. Only one connected component is checked.
2. The loop

   ```
   for each (v,w)∈edges(G) do
   ```

   should exclude the neighbour of v from which you just came, so as to prevent a single edge *w-v* from being classified as a cycle.

Computing Connected Components

Problems:

- how many connected subgraphs are there?
- are two vertices in the same connected subgraph?

Both of the above can be solved if we can

- build an array, one element for each vertex V
- indicating which connected component V is in

- `componentOf[]` … array [0..nV-1] of component IDs

... Computing Connected Components

Algorithm to assign vertices to connected components:

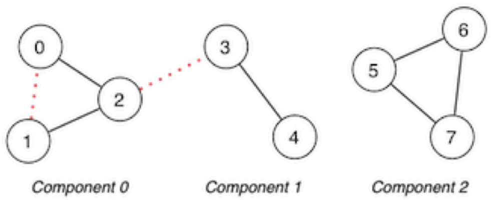
```
components(G):
|  Input graph G
|
|  for all vertices v∈G do
|     componentOf[v]=-1
|  end for
|  compID=0
|  for all vertices v∈G do
|  |  if componentOf[v]=-1 then
|  |     dfsComponents(G,v,compID)
|  |     compID=compID+1
|  |  end if
|  end for

dfsComponents(G,v,id):
|  componentOf[v]=id
|  for all vertices w adjacent to v do
|     if componentOf[w]=-1 then
|        dfsComponents(G,w,id)
|     end if
|  end for
```

Exercise #2: Connected components

Trace the execution of the algorithm

1. on the graph shown below
2. on the same graph but with the dotted edges added



Consider neighbours in ascending order

1.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
-1	-1	-1	-1	-1	-1	-1	-1
0	-1	-1	-1	-1	-1	-1	-1
0	-1	0	-1	-1	-1	-1	-1
0	0	0	-1	-1	-1	-1	-1
0	0	0	1	-1	-1	-1	-1
…							
0	0	0	1	1	2	2	2

2.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
-1	-1	-1	-1	-1	-1	-1	-1
0	-1	-1	-1	-1	-1	-1	-1
0	0	-1	-1	-1	-1	-1	-1
0	0	0	-1	-1	-1	-1	-1
…							
0	0	0	0	0	1	1	1

Hamiltonian and Euler Paths

Hamiltonian Path and Circuit

Hamiltonian path problem:

- find a simple path connecting two vertices *v,w* in graph *G*
- such that the path includes each *vertex* exactly once

If $v = w$, then we have a *Hamiltonian circuit*

Simple to state, but difficult to solve (*NP*-complete)

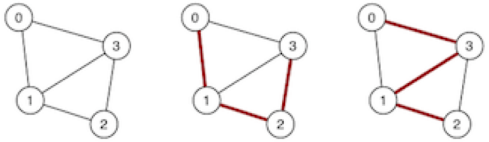
Many real-world applications require you to visit all vertices of a graph:

- Travelling salesman
- Bus routes
- …

Problem named after Irish mathematician, physicist and astronomer Sir William Rowan Hamilton (1805 - 1865)

... Hamiltonian Path and Circuit

Graph and two possible Hamiltonian paths:



... Hamiltonian Path and Circuit

Approach:

- generate all possible simple paths (using e.g. DFS)
- keep a counter of vertices visited in current path
- stop when find a path containing *V* vertices

Can be expressed via a recursive DFS algorithm

- similar to simple path finding approach, except
  - keeps track of path length; succeeds if length = *v*
  - resets "visited" marker after unsuccessful path

... Hamiltonian Path and Circuit

Algorithm for finding Hamiltonian path:

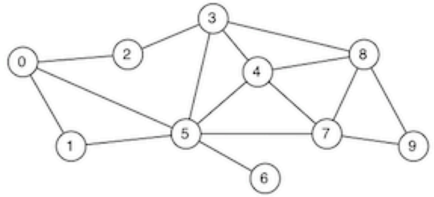
```
visited[] // array [0..nV-1] to keep track of visited vertices

hasHamiltonianPath(G,src,dest):
|  for all vertices v∈G do
|     visited[v]=false
|  end for
|  return hamiltonR(G,src,dest,#vertices(G)-1)

hamiltonR(G,v,dest,d):
|  Input G    graph
|        v    current vertex considered
|        dest destination vertex
|        d    distance "remaining" until path found
|
|  if v=dest then
|     if d=0 then return true else return false
|  else
|  |  visited[v]=true
|  |  for each (v,w)∈edges(G) ∧ ¬visited[w] do
|  |     if hamiltonR(G,w,dest,d-1) then
|  |        return true
|  |     end if
|  |  end for
|  end if
|  visited[v]=false            // reset visited mark
|  return false
```

Exercise #3: Hamiltonian Path

Trace the execution of the algorithm when searching for a Hamiltonian path from 1 to 6:



Consider neighbours in ascending order

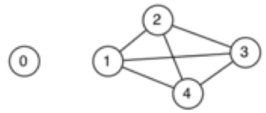
1-0-2-3-4-5-6	d≠0
1-0-2-3-4-5-7-8-9	no unvisited neighbour
1-0-2-3-4-5-7-9-8	no unvisited neighbour
1-0-2-3-4-7-5-6	d≠0
1-0-2-3-4-7-8-9	no unvisited neighbour
1-0-2-3-4-7-9-8	no unvisited neighbour
1-0-2-3-4-8-7-5-6	d≠0
1-0-2-3-4-8-7-9	no unvisited neighbour
1-0-2-3-4-8-9-7-5-6	✓

Repeat on your own with `src=0` and `dest=6`

... Hamiltonian Path and Circuit

Analysis: worst case requires *(V-1)!* paths to be examined

Consider a graph with isolated vertex and the rest fully-connected



Checking `hasHamiltonianPath(g,`*x*`,0)` for any *x*

- requires us to consider every possible path
- e.g 1-2-3-4, 1-2-4-3, 1-3-2-4, 1-3-4-2, 1-4-2-3, …
- starting from any *x*, there are 3! paths ⇒ 4! total paths
- there is no path of length 5 in these *(V-1)!* possibilities

There is no known simpler algorithm for this task ⇒ *NP*-hard.

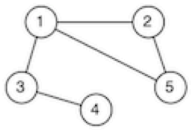
Note, however, that the above case could be solved in constant time if we had a fast check for 0 and *x* being in the same connected component

Euler Path and Circuit

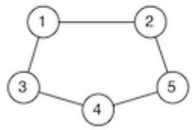
Euler path problem:

- find a path connecting two vertices *v*,*w* in graph *G*
- such that the path includes each *edge* exactly once
  (note: the path does not have to be simple ⇒ can visit vertices more than once)

If $v = w$, the we have an *Euler circuit*



Euler Path: 4-3-1-5-2-1



Euler Circuit: 1-2-5-4-3-1

Many real-world applications require you to visit all edges of a graph:

- Postman
- Garbage pickup
- …

Problem named after Swiss mathematician, physicist, astronomer, logician and engineer Leonhard Euler (1707 - 1783)

... Euler Path and Circuit

One possible "brute-force" approach:

- check for each path if it's an Euler path
- would result in factorial time performance

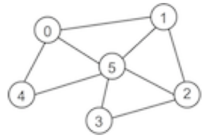
Can develop a better algorithm by exploiting:

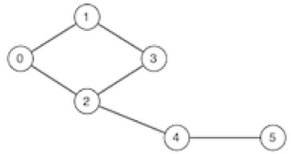
Theorem. *A graph has an Euler circuit* if and only if
it is connected and all vertices have even degree

Theorem. *A graph has a non-circuitous Euler path* if and only if
it is connected and exactly two vertices have odd degree

Exercise #4: Euler Paths and Circuits

Which of these two graphs have an Euler path? an Euler circuit?





No Euler circuit

Only the second graph has an Euler path, e.g. 2-0-1-3-2-4-5

... Euler Path and Circuit

Assume the existence of `degree(g,v)` (degree of a vertex, cf. problem set week 6)

Algorithm to check whether a graph has an Euler path:

```
hasEulerPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if G has Euler path from src to dest
|         false otherwise
|
|  if src≠dest then
|     if degree(G,src) or degree(G,dest) is even then
|        return false
|     end if
|  else if degree(G,src) is odd then
|     return false
|  end if
|  for all vertices v∈G do
|     if v≠src and v≠dest and degree(G,v) is odd then
|        return false
|     end if
|  end for
|  return true
```

... Euler Path and Circuit

Analysis of `hasEulerPath` algorithm:

- assume that connectivity is already checked
- assume that degree is available via *O(1)* lookup
- single loop over all vertices ⇒ *O(V)*

If degree requires iteration over vertices

- cost to compute degree of a single vertex is $O(V)$
- overall cost is $O(V^2)$

⇒ problem tractable, even for large graphs (unlike Hamiltonian path problem)

For the keen, a linear-time (in the number of edges, E) algorithm to compute an Euler path is described in [Sedgewick] Ch.17.7.

Directed Graphs

Directed Graphs (Digraphs)

In our previous discussion of graphs:

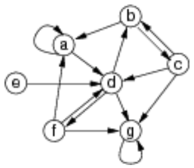
- an edge indicates a relationship between two vertices
- an edge indicates nothing more than a relationship

In many real-world applications of graphs:

- edges are directional ($v \rightarrow w \neq w \rightarrow v$)
- edges have a *weight* (cost to go from $v \rightarrow w$)

... Directed Graphs (Digraphs)

Example digraph and adjacency matrix representation:



	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

Undirectional ⇒ symmetric matrix
Directional ⇒ non-symmetric matrix

Maximum #edges in a digraph with V vertices: V^2

... Directed Graphs (Digraphs)

Terminology for digraphs …

Directed path: sequence of $n \geq 2$ vertices $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_n$

- where $(v_i,v_{i+1}) \in edges(G)$ for all v_i,v_{i+1} in sequence
- if $v_1 = v_n$, we have a *directed cycle*

Reachability: w is reachable from v if ∃ directed path $v,\ldots,w$

Digraph Applications

Potential application areas:

Domain	**Vertex**	**Edge**
Web	web page	hyperlink
scheduling	task	precedence
chess	board position	legal move
science	journal article	citation
dynamic data	malloc'd object	pointer
programs	function	function call
`make`	file	dependency

... Digraph Applications

Problems to solve on digraphs:

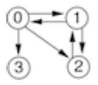
- is there a directed path from s to t? (transitive closure)
- what is the shortest path from s to t? (shortest path)
- are all vertices mutually reachable? (strong connectivity)
- how to organise a set of tasks? (topological sort)
- which web pages are "important"? (PageRank)
- how to build a web crawler? (graph traversal)

Digraph Representation

Similar set of choices as for undirectional graphs:

- array of edges (directed)
- vertex-indexed adjacency matrix (non-symmetric)
- vertex-indexed adjacency lists

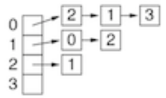
V vertices identified by 0 .. V-1



digraph

	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	0	1	0	0
3	0	0	0	0

adj matrix



adj lists

Reachability

Transitive Closure

33/51

Given a digraph G it is potentially useful to know

- is vertex *t* reachable from vertex *s*?

Example applications:

- can I complete a schedule from the current state?
- is a malloc'd object being referenced by any pointer?

How to compute transitive closure?

... Transitive Closure

34/51

One possibility:

- implement it via `hasPath(G,s,t)` (itself implemented by DFS or BFS algorithm)
- feasible if *reachable(G,s,t)* is infrequent operation

What if we have an algorithm that frequently needs to check reachability?

Would be very convenient/efficient to have:

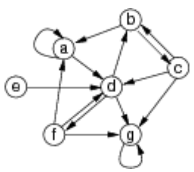
```
reachable(G,s,t):
|  return G.tc[s][t]   // transitive closure matrix
```

Of course, if *V* is *very* large, then this is not feasible.

Exercise #5: Transitive Closure Matrix

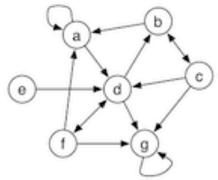
35/51

Which reachable *s* .. *t* exist in the following graph?



	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

Transitive closure of example graph:



	a	b	c	d	e	f	g
a	1	0	0	1	0	0	0
b	1	0	1	0	0	0	0
c	0	1	0	1	0	0	1
d	0	1	0	0	0	1	1
e	0	0	0	1	0	0	0
f	1	0	0	1	0	0	1
g	0	0	0	0	0	0	1

adjacency matrix

	a	b	c	d	e	f	g
a	1	1	1	1	0	1	1
b	1	1	1	1	0	1	1
c	1	1	1	1	0	1	1
d	1	1	1	1	0	1	1
e	1	1	1	1	0	1	1
f	1	1	1	1	0	1	1
g	0	0	0	0	0	0	1

reachability matrix

... Transitive Closure

37/51

Goal: produce a matrix of reachability values

- if *tc[s][t]* is 1, then *t* is reachable from *s*
- if *tc[s][t]* is 0, then *t* is not reachable from *s*

So, how to create this matrix?

Observation:

$\forall i,s,t \in$ vertices(G):
  $(s,i) \in$ edges(G) and $(i,t) \in$ edges(G) $\Rightarrow$ $tc[s][t] = 1$

`tc[s][t]=1` if there is a path from *s* to *t* of length 2 ($s \to i \to t$)

... Transitive Closure

38/51

If we implement the above as:

```
make tc[][] a copy of edges[][]
for all i∈vertices(G) do
   for all s∈vertices(G) do
      for all t∈vertices(G) do
         if tc[s][i]=1 and tc[i][t]=1 then
            tc[s][t]=1
         end if
      end for
   end for
end for
```

then we get an algorithm to convert `edges` into a *tc*

This is known as *Warshall's algorithm*

... Transitive Closure

39/51

How it works …

After iteration 1, `tc[s][t]` is 1 if

- either $s \rightarrow t$ exists or $s \rightarrow 0 \rightarrow t$ exists

After iteration 2, `tc[s][t]` is 1 if any of the following exist

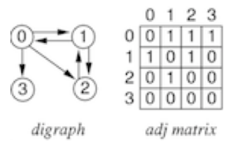
- $s \rightarrow t$ or $s \rightarrow 0 \rightarrow t$ or $s \rightarrow 1 \rightarrow t$ or $s \rightarrow 0 \rightarrow 1 \rightarrow t$ or $s \rightarrow 1 \rightarrow 0 \rightarrow t$

Etc. … so after the V^{th} iteration, `tc[s][t]` is 1 if

- there is any directed path in the graph from s to t

Exercise #6: Transitive Closure

Trace Warshall's algorithm on the following graph:



1st iteration `i=0`:

tc	[0]	[1]	[2]	[3]
[0]	0	1	1	1
[1]	1	1	1	1
[2]	0	1	0	0
[3]	0	0	0	0

2nd iteration `i=1`:

tc	[0]	[1]	[2]	[3]
[0]	1	1	1	1
[1]	1	1	1	1
[2]	1	1	1	1
[3]	0	0	0	0

3rd iteration `i=2`: unchanged

4th iteration `i=3`: unchanged

... Transitive Closure

Cost analysis:

- storage: additional V^2 items (each item may be 1 bit)
- computation of transitive closure: V^3
- computation of `reachable()`: $O(1)$ after having generated `tc[][]`

Amortisation: would need many calls to `reachable()` to justify other costs

Alternative: use DFS in each call to `reachable()`
Cost analysis:

- storage: cost of queue and set during reachable
- computation of `reachable()`: cost of DFS = $O(V^2)$ (for adjacency matrix)

Digraph Traversal

Same algorithms as for undirected graphs:

`depthFirst(v):`

1. mark `v` as visited
2. for each `(v,w)`∈*edges*(G) do
   if `w` has not been visited then
   `depthFirst(w)`

`breadth-first(v):`

1. enqueue `v`
2. while queue not empty do
   dequeue `v`
   if `v` not already visited then
   mark `v` as visited
   enqueue each vertex `w` adjacent to `v`

Example: Web Crawling

Goal: visit every page on the web

Solution: breadth-first search with "implicit" graph

```
webCrawl(startingURL):
|  mark startingURL as alreadySeen
|  enqueue(Q,startingURL)
|  while Q is not empty do
|  |  nextPage=dequeue(Q)
|  |  visit nextPage
|  |  for each hyperLink on nextPage do
|  |  |  if hyperLink not alreadySeen then
|  |  |     mark hyperLink as alreadySeen
|  |  |     enqueue(Q,hyperLink)
|  |  |  end if
|  |  end for
```

```
|   end while
```

`visit` scans page and collects e.g. keywords and links

PageRank

Goal: determine which Web pages are "important"

Approach: ignore page contents; focus on hyperlinks

- treat Web as graph: page = vertex, hyperlink = di-edge
- pages with many incoming hyperlinks are important
- need to computing "incoming degree" for vertices

Problem: the Web is a *very* large graph

- approx. 10^{14} pages, 10^{15} hyperlinks

Assume for the moment that we could build a graph …

Most frequent operation in algorithm "Does edge (v,w) exist?"

... PageRank

Simple PageRank algorithm:

```
PageRank(myPage):
|   rank=0
|   for each page in the Web do
|   |   if linkExists(page,myPage) then
|   |      rank=rank+1
|   |   end if
|   end for
```

Note: requires *inbound* link check (not outbound as assumed above for cost of representation)

... PageRank

V = # pages in Web, E = # hyperlinks in Web

Costs for computing PageRank for each representation:

Representation	linkExists(v,w)	Cost
Adjacency *matrix*	`edge[v][w]`	1
Adjacency *lists*	`inLL(list[v],w)`	$\cong E/V$

Not feasible ...

- adjacency matrix ... $V \cong 10^{14} \Rightarrow$ matrix has 10^{28} cells
- adjacency list ... V lists, each with $\cong$10 hyperlinks $\Rightarrow 10^{15}$ list nodes

So how to really do it?

... PageRank

Approach: the random web surfer

- if we randomly follow links in the web …
- … more likely to re-discover pages with many inbound links

```
curr=random page, prev=null
for a long time do
|   if curr not in array ranked[] then
|      rank[curr]=0
|   end if
|   rank[curr]=rank[curr]+1
|   if random(0,100)<85 then              // with 85% chance ...
|      prev=curr
|      curr=choose hyperlink from curr    // ... crawl on
|   else
|      curr=random page                   // avoid getting stuck
|      prev=null
|   end if
end for
```

Could be accomplished while we crawl web to build search index

---

Exercise #7: Implementing Facebook

Facebook could be considered as a giant "social graph"

- what are the vertices?
- what are the edges?
- are edges directional?

What kind of algorithm would …

- help us find people that you might like to "befriend"?

Tips for Week 7 Problem Set

Main theme: *Graph traversal, digraphs*

- Test your understanding of Euler/Hamiltonian paths/cycles
- Test your understanding of directed graphs

- Algorithms:
  - correct the "buggy" cycle check from above
  - maintain "connected component array" as part of graph ADT implementation

- *Do the online mock test*
- *Review all concepts, data structures, algorithms from weeks 2-7*

Summary

- Graph traversal: cycle check, connected components
- Hamiltonian paths/circuits, Euler paths/circuits
- Digraphs: representations, applications
- Warshall's algorithm to compute reachability

- Suggested reading (Sedgewick):
  - Hamiltonian/Euler paths … Ch.17.7
  - Digraphs … Ch.19.1-19.3
