Week 3: Graph Data Structures and Algorithms

Graph Definitions

Graphs

6/188

Many applications require

- a collection of *items* (i.e. a set)
- *relationships*/connections between items

Examples:

- maps: items are cities, connections are roads
- web: items are pages, connections are hyperlinks

Collection types you're familiar with

- arrays and lists … linear sequence of items

Graphs are more general … allow arbitrary connections

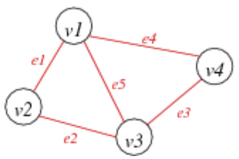
... Graphs

7/188

A *graph G = (V,E)*

- *V* is a set of *vertices*
- *E* is a set of *edges* (subset of $V \times V$)

Example:



V = {v1, v2, v3, v4}

E = {e1, e2, e3, e4, e5}

... Graphs

8/188

A real example: Australian road distances

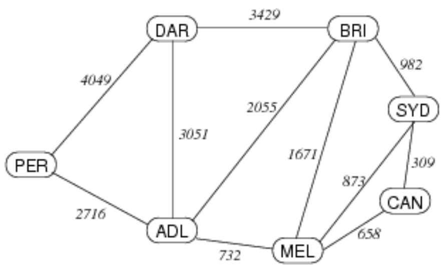
Distance	Adelaide	Brisbane	Canberra	Darwin	Melbourne	Perth	Sydney
Adelaide	-	2055	-	3051	732	2716	-
Brisbane	2055	-	-	3429	1671	-	982
Canberra	-	-	-	-	658	-	309
Darwin	3051	3429	-	-	-	4049	-
Melbourne	732	1671	658	-	-	-	873
Perth	2716	-	-	4049	-	-	-
Sydney	-	982	309	-	873	-	-

Notes: vertices are cities, edges are distance between cities, symmetric

... Graphs

9/188

Alternative representation of above:



... Graphs

10/188

Questions we might ask about a graph:

- is there a way to get from item A to item B?
- what is the best way to get from A to B?
- which items are connected?

Graph algorithms are generally more complex than linked list ones:

- no implicit order of items
- graphs may contain cycles
- concrete representation is less obvious
- algorithm complexity depends on connection complexity

Properties of Graphs

11/188

Terminology: *|V|* and *|E|* (cardinality) normally written just as *V* and *E*.

A graph with *V* vertices has at most *V(V-1)/2* edges.

The ratio *E:V* can vary considerably.

- if *E* is closer to V^2, the graph is *dense*
- if *E* is closer to *V*, the graph is *sparse*
    - Example: web pages and hyperlinks, intersections and roads on street map

Knowing whether a graph is sparse or dense is important

- may affect choice of data structures to represent graph

- may affect choice of algorithms to process graph

Exercise #1: Number of Edges

12/188

The edges in a graph represent pairs of connected vertices. A graph with V has V^2 such pairs.

Consider $V = \{1,2,3,4,5\}$ with all possible pairs:

$E = \{ (1,1), (1,2), (1,3), (1,4), (1,5), (2,1), (2,2), \ldots, (4,5), (5,5) \}$

Why do we say that the maximum #edges is $V(V-1)/2$?

… because

- (v,w) and (w,v) denote the same edge (in an undirected graph)
- we do not consider loops (v,v) (in undirected graphs)

Graph Terminology

14/188

For an edge e that connects vertices v and w

- v and w are *adjacent* (neighbours)
- e is *incident* on both v and w

Degree of a vertex v

- number of edges incident on v

Synonyms:

- vertex = node, edge = arc = link (NB: some people use arc for *directed* edges)

... Graph Terminology

15/188

Path: a sequence of vertices where

- each vertex has an edge to its predecessor

Simple path: a path where

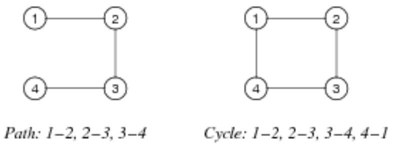
- all vertices and edges are different

Cycle: a path

- that is simple except last vertex = first vertex

Length of path or cycle:

- #edges



... Graph Terminology

16/188

Connected graph

- there is a *path* from each vertex to every other vertex
- if a graph is not connected, it has ≥2 *connected components*

Complete graph K_V

- there is an *edge* from each vertex to every other vertex
- in a complete graph, $E = V(V-1)/2$

... Graph Terminology

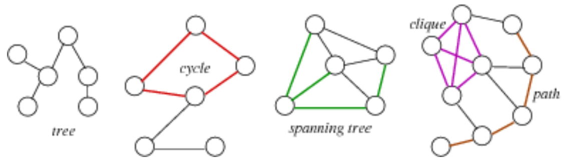
17/188

Tree: connected (sub)graph with no cycles

Spanning tree: tree containing all vertices

n-Clique: complete subgraph on n nodes

Consider the following single graph:



This graph has 26 vertices, 33 edges, and 4 connected components

Note: The entire graph has no spanning tree; what is shown in green is a spanning tree of the third connected component

... Graph Terminology

18/188

A *spanning tree* of connected graph $G = (V,E)$

- is a subgraph of G containing all of V
- and is a single tree (connected, no cycles)

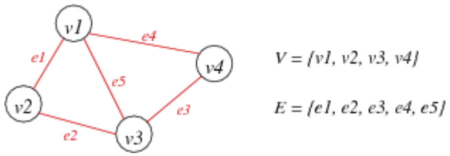
A *spanning forest* of non-connected graph $G = (V,E)$

- is a subgraph of G containing all of V

- and is a set of trees (not connected, no cycles),
  - with one tree for each *connected component*

Exercise #2: Graph Terminology

19/188



1. How many edges need to be removed to obtain a spanning tree?
2. How many different spanning trees?

---

1. 2
2. $\frac{5 \cdot 4}{2} - 2 = 8$ spanning trees (no spanning tree if we remove *{e1,e2}* or *{e3,e4}*)

... Graph Terminology

21/188

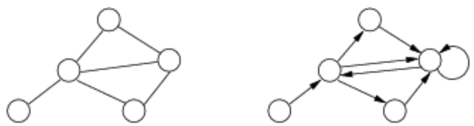
Undirected graph

- *edge(u,v) = edge(v,u)*, no self-loops (i.e. no *edge(v,v)*)

Directed graph

- *edge(u,v) ≠ edge(v,u)*, can have self-loops (i.e. *edge(v,v)*)

  Example:



Undirected graph *Directed graph*

Weighted graph

- each edge has an associated value (weight)
- e.g. road map (weights on edges are distances between cities)

Other types of graphs …

Multi-graph

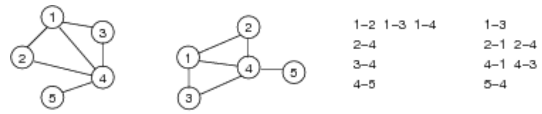
- allow multiple edges between two vertices
- e.g. function call graph (`f()` calls `g()` in several places)

Graph Data Structures

Graph Representations

23/188

Four representations of the *same* graph:



(a) *(b)* *(c)* *(d)*

We will discuss three different graph data structures:

1. Array of edges
2. Adjacency matrix
3. Adjacency list

Array-of-edges Representation

24/188

Edges are represented as an array of `Edge` values (= pairs of vertices)

- disadvantage: deleting edges is slightly complex
- undirected: order of vertices in an `Edge` *(v,w)* doesn't matter

[(0,1), (1,2), (1,3), (2,3)]   [(1,0), (1,1), (0,2), (0,3), (2,3)]

For simplicity, we always assume vertices to be numbered `0..V-1`

... Array-of-edges Representation

25/188

Graph initialisation

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V   // #vertices (numbered 0..V-1)
|  g.nE = 0   // #edges
|  allocate enough memory for g.edges[]
|  return g
```

How much is enough? … No more than *V(V-1)/2* … Much less in practice (sparse graph)

... Array-of-edges Representation

26/188

Edge insertion

```
insertEdge(g,(v,w)):
```

```
|  Input graph g, edge (v,w)  // assumption: (v,w) not in g
|
|  g.edges[g.nE]=(v,w)
|  g.nE=g.nE+1
```

Edge removal

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)  // assumption: (v,w) in g
|
|  i=0
|  while (v,w)≠g.edges[i] do
|     i=i+1
|  end while
|  g.edges[i]=g.edges[g.nE-1] // replace (v,w) by last edge in array
|  g.nE=g.nE-1
```


Cost Analysis

Storage cost: *O(E)*

Cost of operations:

- initialisation: *O(1)*
- insert edge: *O(1)* (assuming edge array has space)
- find/delete edge: *O(E)* (need to find edge in edge array)

If array is full on insert

- allocate space for a bigger array, copy edges across ⇒ *O(E)*

If we maintain edges in order

- use binary search to insert/find edge ⇒ *O(log E)*
  (requires binary search *tree* of edges → week 4)

Exercise #3: Array-of-edges Representation

Assuming an array-of-edges representation …

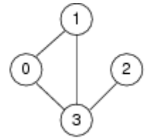
Write an algorithm to output all edges of the graph

```
show(g):
|  Input graph g
|
|  for all i=0 to g.nE-1 do
|     print g.edges[i]
|  end for
```

Time complexity: *O(E)*

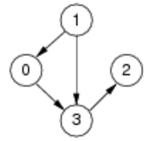
Adjacency Matrix Representation

Edges represented by a *V × V* matrix



Undirected graph

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



Directed graph

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

- easily implemented as 2-dimensional array
- can represent graphs, digraphs and weighted graphs
  - graphs: symmetric boolean matrix
  - digraphs: non-symmetric boolean matrix
  - weighted: non-symmetric matrix of weight values
- disadvantage: if few edges (sparse) ⇒ memory-inefficient

... Adjacency Matrix Representation

Graph initialisation

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V   // #vertices (numbered 0..V-1)
|  g.nE = 0   // #edges
|  allocate memory for g.edges[][]
|  for all i,j=0..V-1 do
|     g.edges[i][j]=0   // false
|  end for
|  return g
```


... Adjacency Matrix Representation

Edge insertion

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]=0 then  // (v,w) not in graph
|     g.edges[v][w]=1       // set to true
|     g.edges[w][v]=1
|     g.nE=g.nE+1
|  end if
```

Edge removal

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  if g.edges[v][w]≠0 then  // (v,w) in graph
|     g.edges[v][w]=0       // set to false
|     g.edges[w][v]=0
|     g.nE=g.nE-1
|  end if
```

Exercise #4: Adjacency-matrix Representation

33/188

Assuming an adjacency matrix representation …

Write an algorithm to output all edges of the graph (no duplicates!)

... Adjacency Matrix Representation

34/188

```
show(g):
|  Input graph g
|
|  for all i=0 to g.nV-2 do
|  |  for all j=i+1 to g.nV-1 do
|  |     if g.edges[i][j] then
|  |        print i"—"j
|  |     end if
|  |  end for
|  end for
```

Time complexity: $O(V^2)$

Exercise #5:

35/188

Analyse storage cost and time complexity of adjacency matrix representation

---

Storage cost: $O(V^2)$

If the graph is sparse, most storage is wasted.

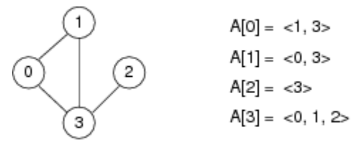
Cost of operations:

- initialisation: $O(V^2)$ (initialise $V \times V$ matrix)
- insert edge: $O(1)$ (set two cells in matrix)
- delete edge: $O(1)$ (unset two cells in matrix)

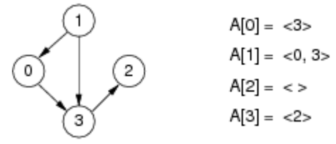
Adjacency List Representation

37/188

For each vertex, store linked list of adjacent vertices:



Undirected graph



Directed graph

- relatively easy to implement in languages like C
- can represent graphs and digraphs
- memory efficient if $E:V$ relatively small

- disadvantage: one graph has many possible representations
  (unless lists are ordered by same criterion e.g. ascending)

... Adjacency List Representation

38/188

Graph initialisation

```
newGraph(V):
|  Input  number of nodes V
|  Output new empty graph
|
|  g.nV = V   // #vertices (numbered 0..V-1)
|  g.nE = 0   // #edges
|  allocate memory for g.edges[]
|  for all i=0..V-1 do
|     g.edges[i]=NULL   // empty list
|  end for
|  return g
```

... Adjacency List Representation

39/188

Edge insertion:

```
insertEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
|  insertLL(g.edges[v],w)
|  insertLL(g.edges[w],v)
|  g.nE=g.nE+1
```

Edge removal:

```
removeEdge(g,(v,w)):
|  Input graph g, edge (v,w)
|
```

```
|  deleteLL(g.edges[v],w)
|  deleteLL(g.edges[w],v)
|  g.nE=g.nE-1
```

Exercise #6:

Analyse storage cost and time complexity of adjacency list representation

Storage cost: $O(V+E)$ (V list pointers, total of $2 \cdot E$ list elements)

- the larger of V,E determines the complexity

Cost of operations:

- initialisation: $O(V)$ (initialise V lists)
- insert edge: $O(1)$ (insert one vertex into list)
  - if you don't check for duplicates
- find/delete edge: $O(V)$ (need to find vertex in list)

Comparison of Graph Representations

	array of edges	adjacency matrix	adjacency list
space usage	E	V^2	$V+E$
initialise	1	V^2	V
insert edge	1	1	1
find/delete edge	E	1	V

Other operations:

	array of edges	adjacency matrix	adjacency list
disconnected(v)?	E	V	1
isPath(x,y)?	$E \cdot log\ V$	V^2	$V+E$
copy graph	E	V^2	$V+E$
destroy graph	1	V	$V+E$

Graph Abstract Data Type

Graph ADT

Data:

- set of edges, set of vertices

Operations:

- building: create graph, add edge
- deleting: remove edge, drop whole graph
- scanning: check if graph contains a given edge

Things to note:

- set of vertices is fixed when graph initialised
- we treat vertices as `int`s, but could be arbitrary `Item`s (e.g. individual profiles on a social network)

... Graph ADT

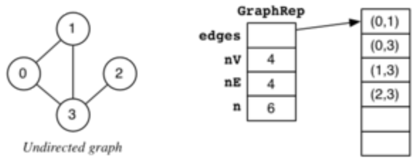
Graph ADT interface **`Graph.h`**

```
// graph representation is hidden
typedef struct GraphRep *Graph;

// vertices denoted by integers 0..N-1
typedef int Vertex;

// edges are pairs of vertices (end-points)
typedef struct Edge { Vertex v; Vertex w; } Edge;

// operations on graphs
Graph newGraph(int V);                  // new graph with V vertices
int   numOfVertices(Graph);             // get number of vertices in a graph
void  insertEdge(Graph, Edge);
void  removeEdge(Graph, Edge);
bool  adjacent(Graph, Vertex, Vertex); /* is there an edge
                                           between two vertices */
void  showGraph(Graph);                 // print all edges in a graph
void  freeGraph(Graph);
```

Graph ADT (Array of Edges)

Implementation of `GraphRep` (array-of-edges representation)

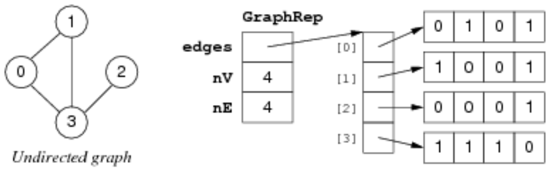
```
typedef struct GraphRep {
   Edge *edges; // array of edges
   int   nV;    // #vertices (numbered 0..nV-1)
   int   nE;    // #edges
   int   n;     // size of edge array
} GraphRep;
```



Graph ADT (Adjacency Matrix)

Implementation of `GraphRep` (adjacency-matrix representation)

```
typedef struct GraphRep {
   int  **edges; // adjacency matrix
   int    nV;    // #vertices
   int    nE;    // #edges
} GraphRep;
```



Undirected graph

... Graph ADT (Adjacency Matrix)

Implementation of graph initialisation (adjacency-matrix representation)

```
Graph newGraph(int V) {
   assert(V >= 0);
   int i;

   Graph g = malloc(sizeof(GraphRep));      assert(g != NULL);
   g->nV = V;  g->nE = 0;

   // allocate memory for each row
   g->edges = malloc(V * sizeof(int *));    assert(g->edges != NULL);
   // allocate memory for each column and initialise with 0
   for (i = 0; i < V; i++) {
      g->edges[i] = calloc(V, sizeof(int)); assert(g->edges[i] != NULL);
   }
   return g;
}
```

standard library function **`calloc(size_t nelems, size_t nbytes)`**

- allocates a memory block of size `nelems*nbytes`
- and sets all bytes in that block to *zero*

... Graph ADT (Adjacency Matrix)

Implementation of edge insertion/removal (adjacency-matrix representation)

```
// check if vertex is valid in a graph
bool validV(Graph g, Vertex v) {
   return (g != NULL && v >= 0 && v < g->nV);
}

void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));

   if (!g->edges[e.v][e.w]) {  // edge e not in graph
      g->edges[e.v][e.w] = 1;
      g->edges[e.w][e.v] = 1;
      g->nE++;
   }
}

void removeEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));

   if (g->edges[e.v][e.w]) {   // edge e in graph
      g->edges[e.v][e.w] = 0;
      g->edges[e.w][e.v] = 0;
      g->nE--;
   }
}
```

Exercise #7: Checking Neighbours

Assuming an adjacency-matrix representation …

Implement a function to check whether two vertices are directly connected by an edge

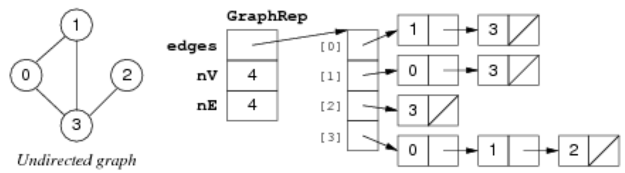
```
bool adjacent(Graph g, Vertex x, Vertex y) { … }
```

```
bool adjacent(Graph g, Vertex x, Vertex y) {
   assert(g != NULL && validV(g,x) && validV(g,y));

   return (g->edges[x][y] != 0);
}
```

Graph ADT (Adjacency List)

Implementation of `GraphRep` (adjacency-list representation)

```
typedef struct GraphRep {
   Node **edges;      // array of lists
   int    nV;         // #vertices
   int    nE;         // #edges
} GraphRep;

typedef struct Node { // linked list node
   Vertex       v;
   struct Node *next;
} Node;
```



Undirected graph

Graph Traversal

Finding a Path

Questions on paths:

- is there a path between two given vertices (*src*,*dest*)?
- what is the sequence of vertices from *src* to *dest*?

Approach to solving problem:

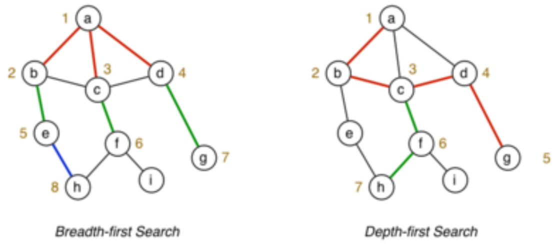
- examine vertices adjacent to *src*
- if any of them is *dest*, then done
- otherwise try vertices two edges from *src*
- repeat looking further and further from *src*

Two strategies for graph traversal/search: *depth-first*, *breadth-first*

- DFS follows one path to completion before considering others
- BFS "fans-out" from the starting vertex ("spreading" subgraph)

... Finding a Path

Comparison of BFS/DFS search for checking if there is a path from *a* to *h* …



Both approaches ignore some edges by remembering previously visited vertices.

Depth-first Search

Depth-first search can be described recursively as

`depthFirst(G,v):`

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

The recursion induces *backtracking*

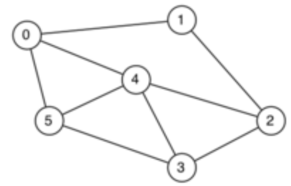
... Depth-first Search

Recursive DFS path checking

```
hasPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if there is a path from src to dest in G,
|         false otherwise
|
|  mark all vertices in G as unvisited
|  return dfsPathCheck(G,src,dest)

dfsPathCheck(G,v,dest):
|  mark v as visited
|  if v=dest then               // found dest
|     return true
|  else
|  |  for all (v,w)∈edges(G) do
|  |  |  if w has not been visited then
|  |  |  |  if dfsPathCheck(G,w,dest) then
|  |  |  |     return true   // found path via w to dest
|  |  |  |  end if
|  |  |  end if
|  |  end for
|  end if
|  return false                 // no path from v to dest
```

Exercise #8: Depth-first Traversal (i)

Trace the execution of `dfsPathCheck(G,0,5)` on:



Consider neighbours in ascending order

Answer:

0 - 1 - 2 - 3 - 4 - 5

... Depth-first Search

Cost analysis:

- all vertices marked as unvisited, each vertex visited at most once ⇒ cost = $O(V)$
- visit all edges incident on visited vertices ⇒ cost = $O(E)$
  - assuming an adjacency list representation

Time complexity of DFS: $O(V+E)$ (adjacency list representation)

- the *larger* of *V,E* determines the complexity

For *dense graphs* … $E \cong V^2 \Rightarrow O(V+E) = O(V^2)$
For *sparse graphs* … $E \cong V \Rightarrow O(V+E) = O(V)$

... Depth-first Search

61/188

Note how different graph data structures affect cost:

- array-of-edges representation
  - visit all edges incident on visited vertices $\Rightarrow$ cost = $O(V \cdot E)$
  - cost of DFS: $O(V \cdot E)$
- adjacency-matrix representation
  - visit all edges incident on visited vertices $\Rightarrow$ cost = $O(V^2)$
  - cost of DFS: $O(V^2)$

... Depth-first Search

62/188

Knowing whether a path exists can be useful

Knowing what the path is even more useful
$\Rightarrow$ record the previously visited node as we search through the graph (so that we can then trace path through graph)

Make use of global variable:

- `visited[]` … array to store previously visited node, for each node being visited

---

... Depth-first Search

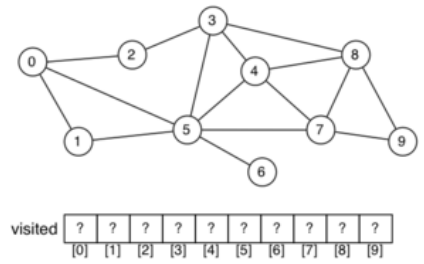
63/188

```
visited[] // store previously visited node, for each vertex 0..nV-1

findPath(G,src,dest):
|  Input graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  visited[src]=src                     // starting node of the path
|  if dfsPathCheck(G,src,dest) then     // show path in dest..src order
|  |  v=dest
|  |  while v≠src do
|  |     print v"-"
|  |     v=visited[v]
|  |  end while
|  |  print src
|  end if

dfsPathCheck(G,v,dest):
|  if v=dest then                 // found edge from v to dest
|     return true
|  else
|  |  for all (v,w)∈edges(G) do
|  |  |  if visited[w]=-1 then
|  |  |  |  visited[w]=v
|  |  |  |  if dfsPathCheck(G,w,dest) then
|  |  |  |  |     return true        // found path via w to dest
|  |  |  |  end if
|  |  |  end if
|  |  end for
|  end if
|  return false                         // no path from v to dest
```

Exercise #9: Depth-first Traversal (ii)

64/188

Show the DFS order in which we visit vertices in this graph when searching for a path from 0 to 6:



Consider neighbours in ascending order

0	**0**	**3**	**5**	**3**	**1**	**5**	**4**	**7**	**8**
[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**

Path: 6-5-1-0

... Depth-first Search

66/188

DFS can also be described non-recursively (via a *stack*):

```
hasPath(G,src,dest):
|  Input  graph G, vertices src,dest
|  Output true if there is a path from src to dest in G,
|         false otherwise
|
|  mark all vertices in G as unvisited
|  push src onto new stack s
|  found=false
|  while not found and s is not empty do
|  |  pop v from s
|  |  mark v as visited
|  |  if v=dest then
|  |     found=true
|  |  else
|  |  |  for each (v,w)∈edges(G) such that w has not been visited
|  |  |     push w onto s
|  |  |  end for
|  |  end if
|  end while
|  return found
```

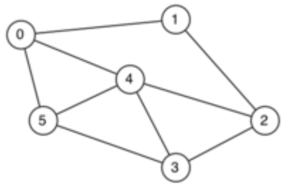
Uses standard stack operations (push, pop, check if empty)

Time complexity is the same: $O(V+E)$

Exercise #10: Depth-first Traversal (iii)

67/188

Show how the stack evolves when executing `findPathDFS(g,0,5)` on:



Push neighbours in *descending* order … so they get popped in ascending order

---

					4	5	
				3	5	5	5
		1	2	4	4	4	4
		4	4	4	4	4	4
(empty)	→ 0	→ 5	→ 5	→ 5	→ 5	→ 5	→ 5

Breadth-first Search

69/188

Basic approach to breadth-first search (BFS):

- visit and mark current vertex
- visit all neighbours of current vertex
- then consider neighbours of neighbours

Notes:

- tricky to describe recursively
- a minor variation on non-recursive DFS search works
  ⇒ switch the *stack* for a *queue*

... Breadth-first Search

70/188

BFS algorithm (records visiting order, marks vertices as visited when put *on* queue):

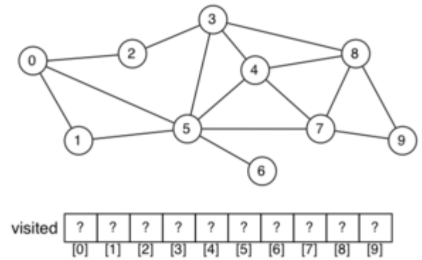
```
visited[] // array of visiting orders, indexed by vertex 0..nV-1

findPathBFS(G,src,dest):
|  Input  graph G, vertices src,dest
|
|  for all vertices v∈G do
|     visited[v]=-1
|  end for
|  enqueue src into new queue q
|  visited[src]=src
|  found=false
|  while not found and q is not empty do
|  |  dequeue v from q
|  |  if v=dest then
|  |     found=true
|  |  else
|  |  |  for each (v,w)∈edges(G) such that visited[w]=-1 do
|  |  |     enqueue w into q
|  |  |     visited[w]=v
|  |  |  end for
|  |  end if
|  end while
|  if found then
|     display path in dest..src order
|  end if
```

Uses standard queue operations (enqueue, dequeue, check if empty)

Exercise #11: Breadth-first Traversal

71/188

Show the state of the queue and the BFS order in which we visit vertices in this graph when searching for a path from 0 to 6:



Consider neighbours in ascending order

Queue (front to the left):

(empty) → 0 → 1 2 5 → 2 5 → 5 3 → 3 4 6 7 → 4 6 7 8 → 6 7 8 → 7 8

0	**0**	**0**	**2**	**5**	**0**	**5**	**5**	**3**	**-1**
[0]	**[1]**	**[2]**	**[3]**	**[4]**	**[5]**	**[6]**	**[7]**	**[8]**	**[9]**

Path: 6-5-0

... Breadth-first Search

73/188

Time complexity of BFS: O(V+E) (adjacency list representation, same as DFS)

BFS finds a "shortest" path

- based on minimum # edges between *src* and *dest*.
- stops with first-found path, if there are multiple ones

In many applications, edges are weighted and we want path

- based on minimum sum-of-weights along path *src* .. *dest*

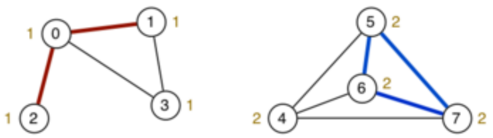
We discuss weighted/directed graphs later.

Other DFS Examples

74/188

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 #12: Buggy Cycle Check

75/188

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

77/188

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

78/188

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
```

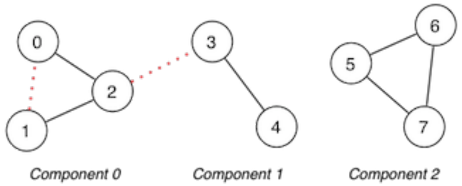
Exercise #13: Connected components

79/188

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 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)

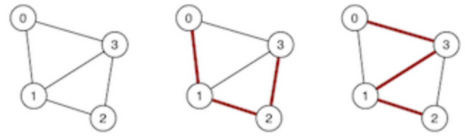
Many real-world applications require you to visit all vertices of a graph:

- 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*-1 (length = *v* for circuit)
  - resets "visited" marker after unsuccessful path

... Hamiltonian Path and Circuit

Algorithm for finding Hamiltonian path:

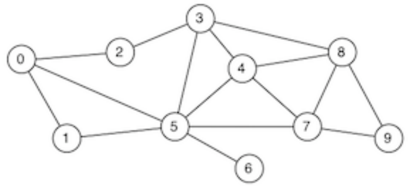
```
visited[] // array [0..nV-1] to keep track of visited vertices

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
|  |  mark v as visited
|  |  for each neighbour w of v in G do
|  |  |  if w has not been visited then
|  |  |     if hamiltonR(G,w,dest,d-1) then
```

```
|   |   |         return true
|   |   |      end if
|   |   |   end if
|   |   end for
|   end if
|   mark v as unvisited             // reset visited mark
|   return false
```

Exercise #14: Hamiltonian Path

86/188

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	✓

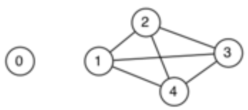
Repeat on your own with `src=0` and `dest=6`

... Hamiltonian Path and Circuit

88/188

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 polynomial algorithm for this task (*NP*-complete)

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

89/188

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

90/188

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:

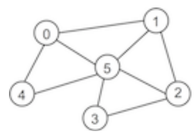
Theorem. *A graph has an Euler circuit* if and only if
it is connected and all vertices have even degree

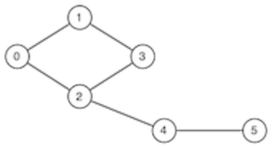
Theorem. *A graph has a non-circuitous Euler path* if and only if
it is connected and exactly two vertices have odd degree

Exercise #15: Euler Paths and Circuits

91/188

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

93/188

Assume the existence of `degree(g,v)` (degree of a vertex, cf. problem set 1 exercise 2 this week)

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       // non-circuitous path
|     if degree(G,src) or degree(G,dest) is even then
|        return false
|     end if
|  else if degree(G,src) is odd then  // circuit
|     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

94/188

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: Linear-time (in the number of edges, *E*) algorithm to compute an Euler path described in [Sedgewick] Ch.17.7

Directed Graphs

Directed Graphs (Digraphs)

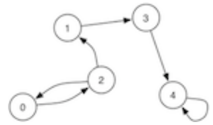
102/188

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 ($v \to w \neq w \to v$)

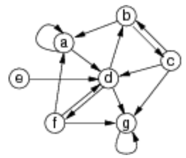


- edges have a *weight* (cost to go from $v \to w$)

... Directed Graphs (Digraphs)

103/188

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)

104/188

Terminology for digraphs …

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

- 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*,…,*w*

Digraph Applications

105/188

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:

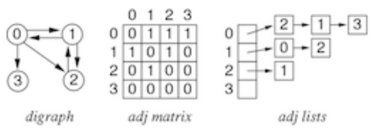
- is there a directed path from *s* to *t*? (transitive closure)
- what is the shortest path from *s* to *t*? (shortest path)
- are all vertices mutually reachable? (strong connectivity)
- how to organise a set of tasks? (topological sort)
- which web pages are "important"? (PageRank)
- how to build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



Reachability

Transitive Closure

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

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?

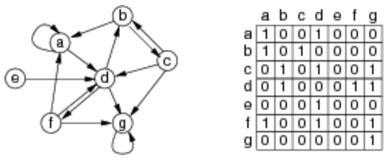
Would be very convenient/efficient to have:

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

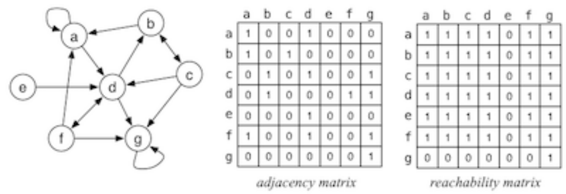
Of course, if *V* is *very* large, then this is not feasible.

Exercise #16: Transitive Closure Matrix

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



Transitive closure of example graph:



... Transitive Closure

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 via some i $(s \to i \to t)$

... Transitive Closure

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

How it works …

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

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

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

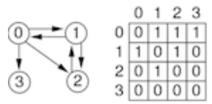
- $s \to t$ or $s \to 0 \to t$ or $s \to 1 \to t$ or $s \to 0 \to 1 \to t$ or $s \to 1 \to 0 \to t$

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 #17: Transitive Closure

Trace Warshall's algorithm on the following graph:



digraph *adj matrix*

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: $O(V^3)$
- 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 stack and set ("visited") 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

120/188

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

Weighted Graphs

Weighted Graphs

122/188

Graphs so far have considered

- edge = an association between two vertices/nodes
- may be a precedence in the association (directed)

Some applications require us to consider

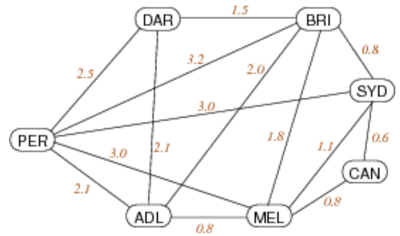
- a *cost* or *weight* of an association
- modelled by assigning values to edges (e.g. positive reals)

Weights can be used in both directed and undirected graphs.

... Weighted Graphs

123/188

Example: major airline flight routes in Australia



Representation: edge = direct flight; weight = approx flying time (hours)

... Weighted Graphs

124/188

Weights lead to minimisation-type questions, e.g.

1. Cheapest way to connect all vertices?

- a.k.a. *minimum spanning tree* problem
- assumes: edges are weighted and undirected

2. Cheapest way to get from *A* to *B*?

- a.k.a *shortest path* problem
- assumes: edge weights positive, directed or undirected

Exercise #18: Implementing a Route Finder

125/188

If we represent a street map as a graph

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

Weighted Graph Representation

126/188

Weights can easily be added to:

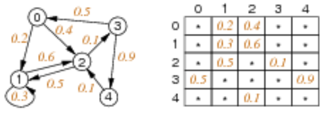
- adjacency matrix representation (0/1 → int or float)
- adjacency lists representation (add int/float to list node)

Both representations work whether edges are directed or not.

... Weighted Graph Representation

127/188

Adjacency matrix representation with weights:



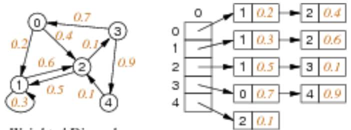
Weighted Digraph *Adjacency Matrix*

Note: need distinguished value to indicate "no edge".

... Weighted Graph Representation

128/188

Adjacency lists representation with weights:



Weighted Digraph

Adjacency Lists

Note: if undirected, each edge appears twice with same weight

---

... Weighted Graph Representation

129/188

Sample adjacency matrix implementation in C requires minimal changes to previous Graph ADT:

WGraph.h

```
// edges are pairs of vertices (end-points) plus positive weight
typedef struct Edge {
   Vertex v;
   Vertex w;
   int    weight;
} Edge;

// returns weight, or 0 if vertices not adjacent
int adjacent(Graph, Vertex, Vertex);
```

... Weighted Graph Representation

130/188

WGraph.c

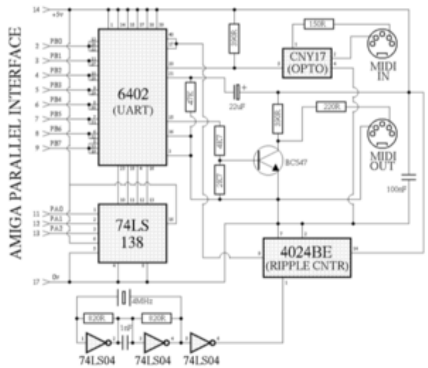
```
typedef struct GraphRep {
   int **edges;  // adjacency matrix storing positive weights
                 // 0 if nodes not adjacent
   int   nV;     // #vertices
   int   nE;     // #edges
} GraphRep;


void insertEdge(Graph g, Edge e) {
   assert(g != NULL && validV(g,e.v) && validV(g,e.w));
   if (g->edges[e.v][e.w] == 0) {  // edge e not in graph
      g->edges[e.v][e.w] = e.weight;
      g->nE++;
   }
}

int adjacent(Graph g, Vertex v, Vertex w) {
   assert(g != NULL && validV(g,v) && validV(g,w));
   return g->edges[v][w];
}
```

Minimum Spanning Trees

Exercise #19: Minimising Wires in Circuits

132/188

Electronic circuit designs often need to make the pins of several components electrically equivalent by wiring them together.



To interconnect a set of n pins we can use an arrangement of n-1 wires each connecting two pins.

What kind of algorithm would …

- help us find the arrangement with the least amount of wire?

Minimum Spanning Trees

133/188

Reminder: *Spanning tree ST* of graph $G=(V,E)$

- *spanning* = all vertices, *tree* = no cycles
  - *ST* is a subgraph of *G* ($G'=(V,E')$ where $E' \subseteq E$)
  - *ST* is *connected* and *acyclic*

Minimum spanning tree MST of graph *G*

- *MST* is a spanning tree of *G*
- sum of edge weights is no larger than any other ST

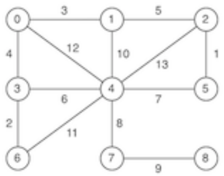
Applications: Computer networks, Electrical grids, Transportation networks …

Problem: how to (efficiently) find MST for graph *G*?

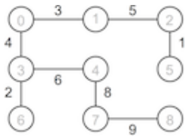
NB: MST may not be unique (e.g. all edges have same weight ⇒ every ST is MST)

... Minimum Spanning Trees

Example:



An MST ...



... Minimum Spanning Trees

Brute force solution:

```
findMST(G):
|  Input  graph G
|  Output a minimum spanning tree of G
|
|  bestCost=∞
|  for all spanning trees t of G do
|  |  if cost(t)<bestCost then
|  |     bestTree=t
|  |     bestCost=cost(t)
|  |  end if
|  end for
|  return bestTree
```

Example of *generate-and-test* algorithm.

Not useful because #spanning trees is potentially large (e.g. n^{n-2} for a complete graph with n vertices)

... Minimum Spanning Trees

Simplifying assumption:

- edges in *G* are not directed (MST for digraphs is harder)

Kruskal's Algorithm

One approach to computing MST for graph *G* with *V* nodes:

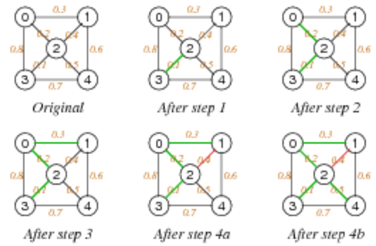
1. start with empty MST
2. consider edges in increasing weight order
   - add edge if it does not form a cycle in MST
3. repeat until *V-1* edges are added

Critical operations:

- iterating over edges in weight order
- checking for cycles in a graph

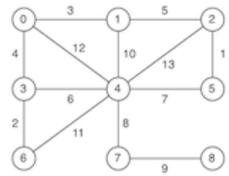
... Kruskal's Algorithm

Execution trace of Kruskal's algorithm:

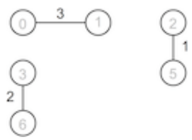


Exercise #20: Kruskal's Algorithm

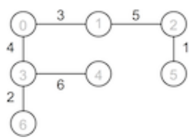
Show how Kruskal's algorithm produces an MST on:



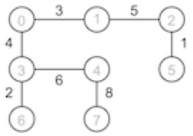
After 3rd iteration:



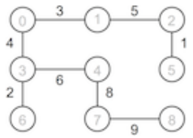
After 6th iteration:



After 7th iteration:



After 8th iteration (V-1=8 edges added):



... Kruskal's Algorithm

Pseudocode:

```
KruskalMST(G):
|  Input  graph G with n nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  sort edges(G) by weight
|  for each e∈sortedEdgeList do
|  |  MST = MST ∪ {e}
|  |  if MST has a cyle then
|  |     MST = MST \ {e}
|  |  end if
|  |  if MST has n-1 edges then
|  |     return MST
|  |  end if
|  end for
```

... Kruskal's Algorithm

Time complexity analysis ...

- sorting edge list is $O(E \cdot log\ E)$
- min V, max E iterations over sorted edges
- on each iteration ...
  - getting next lowest cost edge is $O(1)$
  - checking whether adding it forms a cycle: cost = ??
    - use DFS ... too expensive?
    - could use *Union-Find data structure* (see Sedgewick Ch.1) to maintain sets of connected components
      $\Rightarrow$ loop is $O(E \cdot log\ V)$
- overall complexity $O(E \cdot log\ E) = O(E \cdot log\ V)$

Exercise #21: Kruskal's Algorithm

Why is $O(E \cdot log\ E) = O(E \cdot log\ V)$ in this case?

1. at most $E = V^2$ edges $\Rightarrow$ $log\ E = 2 \cdot log\ V = O(log\ V)$
2. if $V > E+1$ $\Rightarrow$ can ignore all unconnected vertices

Prim's Algorithm

Another approach to computing MST for graph $G=(V,E)$:

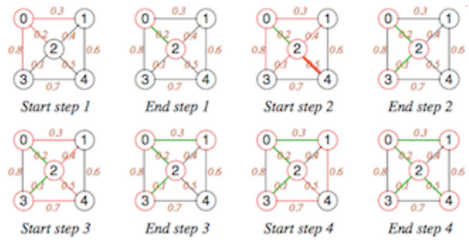
1. start from any vertex v and empty MST
2. choose edge not already in MST to add to MST
   - must be incident on a vertex s already connected to v in MST
   - must be incident on a vertex t not already connected to v in MST
   - must have minimal weight of all such edges
3. repeat until MST covers all vertices

Critical operations:

- checking for vertex being connected in a graph
- finding min weight edge in a set of edges

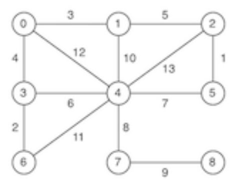
... Prim's Algorithm

Execution trace of Prim's algorithm (starting at s=0):



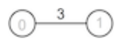
Exercise #22: Prim's Algorithm

Show how Prim's algorithm produces an MST on:

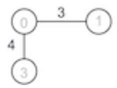


Start from vertex 0

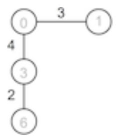
After 1st iteration:



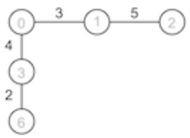
After 2nd iteration:



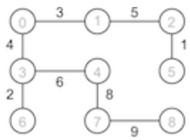
After 3rd iteration:



After 4th iteration:



After 8th iteration (all vertices covered):



... Prim's Algorithm

149/188

Pseudocode:

```
PrimMST(G):
|  Input  graph G with n>0 nodes
|  Output a minimum spanning tree of G
|
|  MST=empty graph
|  usedV={0}
|  unusedE=edges(g)
|  while |usedV|<n do
|  |  find e=(s,t,w)∈unusedE such that {
|  |     s∈usedV, t∉usedV and w is min weight of all such edges
|  |  }
|  |  MST = MST ∪ {e}
|  |  usedV = usedV ∪ {t}
|  |  unusedE = unusedE \ {e}
|  end while
|  return MST
```

Critical operation: finding best edge

... Prim's Algorithm

150/188

Rough time complexity analysis …

- *V* iterations of outer loop
- find min edge with set of edges is *O(E)* ⇒ *O(V·E)* overall

Using a *priority queue* …

- ⇒ *O(E·log V)* overall

Sidetrack: Priority Queues

151/188

Some applications of queues require

- items processed in order of "priority"
- rather than in order of entry (FIFO — first in, first out)

Priority Queues (*PQueues*) provide this via:

- `join`: insert item into PQueue with an associated priority (replacing `enqueue`)
- `leave`: remove item with highest priority (replacing `dequeue`)

Time complexity for naive implementation of a PQueue containing *N* items …

- *O(1)* for `join`   *O(N)* for `leave`

Most efficient implementation ("heap") …

- *O(log N)* for `join`, `leave`   … more on this in week 4 (binary search trees)

Other MST Algorithms

152/188

Boruvka's algorithm … complexity *O(E·log V)*

- the oldest MST algorithm
- start with *V* separate components
- join components using min cost links
- continue until only a single component

Karger, Klein, and Tarjan … complexity *O(E)*

- based on Boruvka, but non-deterministic
- randomly selects subset of edges to consider
- for the keen, here's the paper describing the algorithm

Shortest Path

Shortest Path

154/188

Path = sequence of edges in graph G $p = (v_0,v_1), (v_1,v_2), …, (v_{m-1},v_m)$

cost(path) = sum of edge weights along path

Shortest path between vertices *s* and *t*

- a simple path *p(s,t)* where *s = first(p)*, *t = last(p)*
- no other simple path *q(s,t)* has *cost(q) < cost(p)*

Assumptions: weighted digraph, no negative weights.

Finding shortest path between two given nodes known as *source-target* SP problem

Variations: *single-source* SP, *all-pairs* SP

Applications: navigation, routing in data networks, …

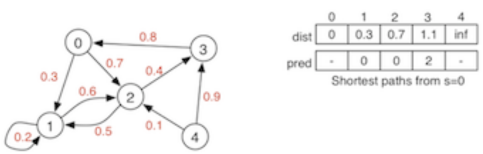
Single-source Shortest Path (SSSP)

Given: weighted digraph *G*, source vertex *s*

Result: shortest paths from *s* to all other vertices

- `dist[]` *V*-indexed array of cost of shortest path from *s*
- `pred[]` *V*-indexed array of predecessor in shortest path from *s*

Example:

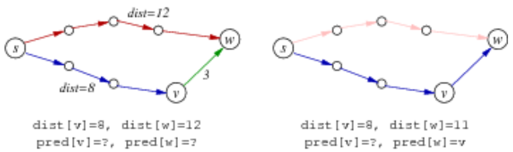


Edge Relaxation

Assume: `dist[]` and `pred[]` as above (but containing data for shortest paths *discovered so far*)

`dist[v]` is length of shortest known path from *s* to *v*
`dist[w]` is length of shortest known path from *s* to *w*

Relaxation updates data for *w* if we find a shorter path from *s* to *w*:



Relaxation along edge *e*=`(v,w,weight)`:

- if **`dist[v]+weight < dist[w]`** then
  update `dist[w]:=dist[v]+weight` and `pred[w]:=v`

Dijkstra's Algorithm

One approach to solving single-source shortest path problem …

Data: *G*, *s*, `dist[]`, `pred[]` and

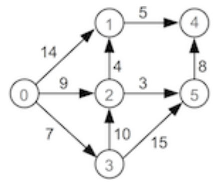
- *vSet*: set of vertices whose shortest path from *s* is unknown

Algorithm:

```
dist[]  // array of cost of shortest path from s
pred[]  // array of predecessor in shortest path from s

dijkstraSSSP(G,source):
|  Input graph G, source node
|
|  initialise dist[] to all ∞, except dist[source]=0
|  initialise pred[] to all -1
|  vSet=all vertices of G
|  while vSet≠∅ do
|  |  find s∈vSet with minimum dist[s]
|  |  for each (s,t,w)∈edges(G) do
|  |     relax along (s,t,w)
|  |  end for
|  |  vSet=vSet\{s}
|  end while
```

Exercise #23: Dijkstra's Algorithm

Show how Dijkstra's algorithm runs on (source node = 0):



	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	∞	∞	∞	∞	∞
pred	–	–	–	–	–	–

---

	[0]	[1]	[2]	[3]	[4]	[5]
dist	0	∞	∞	∞	∞	∞
pred	–	–	–	–	–	–

dist	0	14	9	7	∞	∞
pred	–	0	0	0	–	–

dist	0	14	9	7	∞	22
pred	–	0	0	0	–	3

dist	0	13	9	7	∞	12

pred	–	2	0	0	–	2

dist	0	13	9	7	20	12
pred	–	2	0	0	5	2

dist	0	13	9	7	18	12
pred	–	2	0	0	1	2

... Dijkstra's Algorithm

Why Dijkstra's algorithm is correct:

Hypothesis.
(a) For visited *s* … *dist[s]* is shortest distance from source
(b) For unvisited *t* … *dist[t]* is shortest distance from source *via visited nodes*

Proof.
Base case: no visited nodes, *dist[source]=0*, *dist[s]=∞* for all other nodes

Induction step:

1. If *s* is unvisited node with minimum *dist[s]*, then *dist[s]* is shortest distance from source to *s*:
   - if ∃ shorter path via only visited nodes, then *dist[s]* would have been updated when processing the predecessor of *s* on this path
   - if ∃ shorter path via an unvisited node *u*, then *dist[u]<dist[s]*, which is impossible if *s* has min distance of all unvisited nodes
2. This implies that (a) holds for *s* after processing *s*
3. (b) still holds for all unvisited nodes *t* after processing *s*:
   - if ∃ shorter path via *s* we would have just updated *dist[t]*
   - if ∃ shorter path without *s* we would have found it previously

---

... Dijkstra's Algorithm

Time complexity analysis …

Each edge needs to be considered once ⇒ *O(E)*.

Outer loop has *O(V)* iterations.

Implementing "**find** `s∈vSet` **with** `minimum dist[s]`"

1. try all `s∈vSet` ⇒ cost = $O(V)$ ⇒ overall cost = $O(E + V^2) = O(V^2)$
2. using a PQueue to implement extracting minimum
   - can improve overall cost to $O(E + V \cdot log\ V)$ (for best-known implementation)

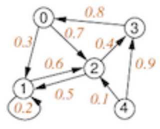
All-pair Shortest Path (APSP)

Given: weighted digraph *G*

Result: shortest paths between all pairs of vertices

- `dist[][]` $V \times V$-indexed matrix of cost of shortest path from v_{row} to v_{col}
- `path[][]` $V \times V$-indexed matrix of next node in shortest path from v_{row} to v_{col}

Example:



Weighted Digraph

V	0	1	2	3	4	
0	0	0.3	0.7	1.1	inf	dist
1	1.8	0	0.6	1.0	inf	
2	1.2	0.5	0	0.4	inf	
3	0.8	1.1	1.5	0	inf	
4	1.3	0.6	0.1	0.5	0	

V	0	1	2	3	4	
0	–	1	2	2	–	path
1	2	–	2	2	–	
2	3	1	–	3	–	
3	0	0	0	–	–	
4	2	2	2	2	–	

Shortest paths between all vertices

Floyd's Algorithm

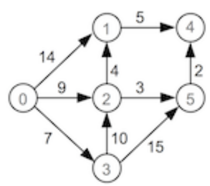
One approach to solving all-pair shortest path problem…

Data: *G*, `dist[][]`, `path[][]` Algorithm:

```
dist[][]  // cost of shortest path from s to t
path[][]  // next node after s on shortest path from s to t

floydAPSP(G):
|  Input graph G
|
|  initialise dist[s][t]=0 for each s=t
|                       =w for each (s,t,w)∈edges(G)
|                       =∞ otherwise
|  initialise path[s][t]=t for each (s,t,w)∈edges(G)
|                       =-1 otherwise
|  for all i∈vertices(G) do
|  |  for all s∈vertices(G) do
|  |  |  for all t∈vertices(G) do
|  |  |  |  if dist[s][i]+dist[i][t] < dist[s][t] then
|  |  |  |     dist[s][t]=dist[s][i]+dist[i][t]
|  |  |  |     path[s][t]=path[s][i]
|  |  |  |  end if
|  |  |  end for
|  |  end for
|  end for
```

Exercise #24: Floyd's Algorithm

Show how Floyd's algorithm runs on:



dist	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	14	9	7		
[1]		0			5	
[2]		4	0			3
[3]			10	0		15
[4]					0	
[5]					2	0

path	[0]	[1]	[2]	[3]	[4]	[5]
[0]		1	2	3		
[1]					4	
[2]		1				5
[3]			2			5
[4]						
[5]					4	

After 1[st] iteration `i=0`: unchanged After 2[nd] iteration `i=1`:

dist	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	14	9	7	19	∞
[1]	∞	0	∞	∞	5	∞
[2]	∞	4	0	∞	9	3
[3]	∞	∞	10	0	∞	15
[4]	∞	∞	∞	∞	0	∞
[5]	∞	∞	∞	∞	2	0

path	[0]	[1]	[2]	[3]	[4]	[5]
[0]	–	1	2	3	1	–
[1]	–	–	–	–	4	–
[2]	–	1	–	–	1	5
[3]	–	–	2	–	–	5
[4]	–	–	–	–	–	–
[5]	–	–	–	–	4	–

After 3[rd] iteration `i=2`:

dist	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	13	9	7	18	12
[1]	∞	0	∞	∞	5	∞
[2]	∞	4	0	∞	9	3
[3]	∞	14	10	0	19	13
[4]	∞	∞	∞	∞	0	∞
[5]	∞	∞	∞	∞	2	0

path	[0]	[1]	[2]	[3]	[4]	[5]
[0]	–	2	2	3	2	2
[1]	–	–	–	–	4	–
[2]	–	1	–	–	1	5
[3]	–	2	2	–	2	2
[4]	–	–	–	–	–	–
[5]	–	–	–	–	4	–

After 4[th] iteration `i=3`: unchanged After 5[th] iteration `i=4`: unchanged After 6[th] iteration `i=5`:

dist	[0]	[1]	[2]	[3]	[4]	[5]
[0]	0	13	9	7	14	12
[1]	∞	0	∞	∞	5	∞
[2]	∞	4	0	∞	5	3
[3]	∞	14	10	0	15	13
[4]	∞	∞	∞	∞	0	∞
[5]	∞	∞	∞	∞	2	0

path	[0]	[1]	[2]	[3]	[4]	[5]
[0]	–	2	2	3	2	2
[1]	–	–	–	–	4	–
[2]	–	1	–	–	5	5
[3]	–	2	2	–	2	2
[4]	–	–	–	–	–	–
[5]	–	–	–	–	4	–

... Floyd's Algorithm

Why Floyd's algorithm is correct:

A shortest path from s to t using only nodes from $\{0,\ldots,i\}$ is the shorter of

- a shortest path from s to t using only nodes from $\{0,\ldots,i\text{-}1\}$
- a shortest path from s to i using only nodes from $\{0,\ldots,i\text{-}1\}$
  plus a shortest path from i to t using only nodes from $\{0,\ldots,i\text{-}1\}$

Also known as Floyd-Warshall algorithm (can you see why?)

... Floyd's Algorithm

Cost analysis …

- initialising `dist[][],path[][]` $\Rightarrow O(V^2)$
- V iterations to update `dist[][],path[][]` $\Rightarrow O(V^3)$

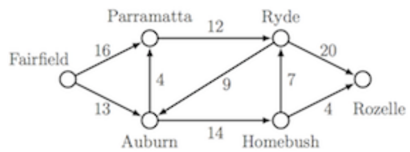
Time complexity of Floyd's algorithm: $O(V^3)$ (same as Warshall's algorithm for transitive closure)

Network Flow

Exercise #25: Merchandise Distribution

Lucky Cricket Company …

- produces cricket balls in Fairfield
- has a warehouse in Rozelle that stocks them
- ships them from factory to warehouse by leasing space on trucks with limited capacity:



What kind of algorithm would …

- help us find the maximum number of crates that can be shipped from Fairfield to Rozelle per day?

Flow Networks

Flow network …

- weighted graph $G=(V,E)$
- distinct nodes $s \in V$ (*source*), $t \in V$ (*sink*)

Edge weights denote *capacities*

Applications:

- Distribution networks, e.g.
  - source: oil field
  - sink: refinery
  - edges: pipes
- Traffic flow

... Flow Networks

171/188

Flow in a network $G=(V,E)$ … nonnegative $f(v,w)$ for all vertices $v,w \in V$ such that

- $f(v,w) \le capacity$ for each edge $e=(v,w,capacity) \in E$
- $f(v,w)=0$ if no edge between v and w
- total flow *into* a vertex = total flow *out of* a vertex:

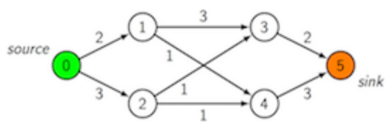
$$\sum_{x \in V} f(x,v) = \sum_{y \in V} f(v,y) \quad \text{for all } v \in V \setminus \{s,t\}$$

Maximum flow … no other flow from s to t has larger value

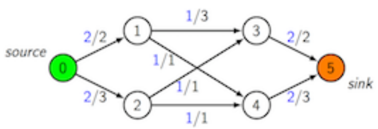
... Flow Networks

172/188

Example:



A (maximum) flow …



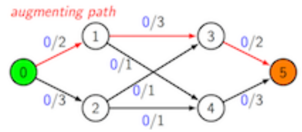
Augmenting Paths

173/188

Assume … $f(v,w)$ contains current flow

Augmenting path: any path from source s to sink t that can currently take more flow

Example:



Residual Network

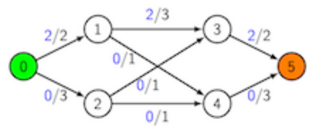
174/188

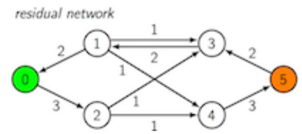
Assume … flow network $G=(V,E)$ and flow $f(v,w)$

Residual network (V,E'):

- same vertex set V
- for each edge $v \xrightarrow{c} w \in E$ …
  - $f(v,w) < c \quad \Rightarrow$ add edge $(v \xrightarrow{c-f(v,w)} w)$ to E'
  - $f(v,w) > 0 \quad \Rightarrow$ add edge $(v \xleftarrow{f(v,w)} w)$ to E'

Example:

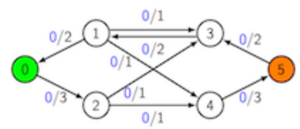




Exercise #26: Augmenting Paths and Residual Networks

175/188

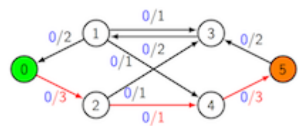
Find an augmenting path in:



and show the residual network after augmenting the flow

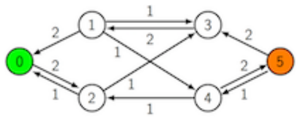
---

1. Augmenting path:



maximum additional flow = 1

2. Residual network:



Can you find a further augmenting path in the new residual network?

Edmonds-Karp Algorithm

One approach to solving maximum flow problem …

maxflow(G):

1. Find a shortest augmenting path
2. Update `flow[][]` so as to represent residual network
3. Repeat until no augmenting path can be found

... Edmonds-Karp Algorithm

Algorithm:

```
flow[][]  // V×V array of current flow
visited[] /* array of predecessor nodes on shortest path
             from source to sink in residual network */

maxflow(G):
|  Input  flow network G with source s and sink t
|  Output maximum flow value
|
|  initialise flow[v][w]=0 for all vertices v, w
|  maxflow=0
|
|  while ∃shortest augmenting path from s to t do  /* Run BFS on "residual network"
|  |                                                   given by capacity[v][w] > flow[v][w]
|  |                                                   to find a shortest path "visited[]" */
|  |  df = maximum additional flow via visited[]
|  |  // adjust flow so as to represent residual graph
|  |  v=t
|  |  while v≠s do
|  |  |  flow[visited[v]][v] = flow[visited[v]][v] + df;
|  |  |  flow[v][visited[v]] = flow[v][visited[v]] - df;
|  |  |  v=visited[v]
|  |  end while
|  |  maxflow=maxflow+df
|  end while
|  return maxflow
```

... Edmonds-Karp Algorithm

Time complexity analysis …

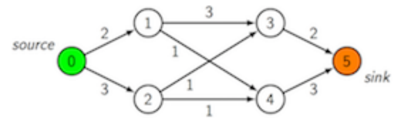
- *Theorem*. The number of augmenting paths needed is at most $V \cdot E/2$.
  $\Rightarrow$ Outer loop has $O(V \cdot E)$ iterations.
- Finding augmenting path $\Rightarrow O(E)$ (consider only vertices connected to source and sink $\Rightarrow O(V+E)=O(E)$)

Overall cost of Edmonds-Karp algorithm: $O(V \cdot E^2)$

Note: Edmonds-Karp algorithm is an implementation of general *Ford-Fulkerson method*

Exercise #27: Edmonds-Karp Algorithm

Show how Edmonds-Karp algorithm runs on:



flow	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]						
[2]						
[3]						
[4]						
[5]						

c>f?	[0]	[1]	[2]	[3]	[4]	[5]
[0]						
[1]						
[2]						
[3]						
[4]						
[5]						

---

flow	[0]	[1]	[2]	[3]	[4]	[5]
[0]	**0**	**0**	**0**	**0**	**0**	**0**
[1]	**0**	**0**	**0**	**0**	**0**	**0**
[2]	**0**	**0**	**0**	**0**	**0**	**0**
[3]	**0**	**0**	**0**	**0**	**0**	**0**
[4]	**0**	**0**	**0**	**0**	**0**	**0**
[5]	**0**	**0**	**0**	**0**	**0**	**0**

c>f?	[0]	[1]	[2]	[3]	[4]	[5]
[0]		✓	✓			
[1]				✓	✓	
[2]				✓	✓	
[3]						✓
[4]						✓
[5]						

df	[0]	[1]	[2]	[3]	[4]	[5]
[0]		**2**	**3**			
[1]				**3**	**1**	
[2]				**1**	**1**	
[3]						**2**
[4]						**3**
[5]						

augmenting path: 0-1-3-5, `df`: 2

flow	[0]	[1]	[2]	[3]	[4]	[5]
[0]	**0**	**2**	**0**	**0**	**0**	**0**
[1]	**-2**	**0**	**0**	**2**	**0**	**0**
[2]	**0**	**0**	**0**	**0**	**0**	**0**
[3]	**0**	**-2**	**0**	**0**	**0**	**2**
[4]	**0**	**0**	**0**	**0**	**0**	**0**
[5]	**0**	**0**	**0**	**-2**	**0**	**0**

c>f?	[0]	[1]	[2]	[3]	[4]	[5]
[0]			✓			
[1]	✓			✓	✓	
[2]				✓	✓	
[3]		✓				
[4]						✓
[5]				✓		

df	[0]	[1]	[2]	[3]	[4]	[5]
[0]			**3**			
[1]	**2**			**1**	**1**	
[2]				**1**	**1**	
[3]		**2**				
[4]						**3**
[5]				**2**		

augmenting path: 0-2-4-5, `df`: 1

`flow`	[0]	[1]	[2]	[3]	[4]	[5]
[0]	**0**	**2**	**1**	**0**	**0**	**0**
[1]	**-2**	**0**	**0**	**2**	**0**	**0**
[2]	**-1**	**0**	**0**	**0**	**1**	**0**
[3]	**0**	**-2**	**0**	**0**	**0**	**2**
[4]	**0**	**0**	**-1**	**0**	**0**	**1**
[5]	**0**	**0**	**0**	**-2**	**-1**	**0**

`c>f?`	[0]	[1]	[2]	[3]	[4]	[5]
[0]			✓			
[1]	✓			✓	✓	
[2]	✓			✓		
[3]		✓				
[4]			✓			✓
[5]				✓	✓	

`df`	[0]	[1]	[2]	[3]	[4]	[5]
[0]			**2**			
[1]	**2**			**1**	**1**	
[2]	**1**			**1**		
[3]		**2**				
[4]			**1**			**2**
[5]				**2**	**1**	

augmenting path: 0-2-3-1-4-5, `df`: 1

`flow`	[0]	[1]	[2]	[3]	[4]	[5]
[0]	**0**	**2**	**2**	**0**	**0**	**0**
[1]	**-2**	**0**	**0**	**1**	**1**	**0**
[2]	**-2**	**0**	**0**	**1**	**1**	**0**
[3]	**0**	**-1**	**-1**	**0**	**0**	**2**
[4]	**0**	**-1**	**-1**	**0**	**0**	**2**
[5]	**0**	**0**	**0**	**-2**	**-2**	**0**

`c>f?`	[0]	[1]	[2]	[3]	[4]	[5]
[0]			✓			
[1]	✓			✓		
[2]	✓					
[3]		✓	✓			
[4]		✓	✓			✓
[5]				✓	✓	

Digraph Applications

PageRank

Goal: determine which Web pages are "important"

Approach: ignore page contents; focus on hyperlinks

- treat Web as graph: page = vertex, hyperlink = directed edge
- pages with many incoming hyperlinks are important
- need to compute "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

... 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 #28: 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"?

Summary

- Graph terminology
  - vertices, edges, vertex degree, connected graph, tree
  - path, cycle, clique, spanning tree, spanning forest
- Graph representations
  - array of edges
  - adjacency matrix
  - adjacency lists
- Graph traversal
  - depth-first search (DFS)
  - breadth-first search (BFS)
  - cycle check, connected components
  - Hamiltonian paths/circuits, Euler paths/circuits
- Digraphs, weighted graphs: representations, applications
- Reachability
  - Warshall
- Minimum Spanning Tree (MST)
  - Kruskal, Prim
- Shortest path problems
  - Dijkstra (single source SPP)
  - Floyd (all-pair SPP)
- Flow networks
  - Edmonds-Karp (maximum flow)

- Suggested reading (Sedgewick):
  - graph representations … Ch. 17.1-17.5
  - Hamiltonian/Euler paths … Ch. 17.7
  - graph search … Ch. 18.1-18.3, 18.7
  - digraphs … Ch. 19.1-19.3
  - weighted graphs … Ch. 20-20.1
  - MST … Ch. 20.2-20.4
  - SSP … Ch. 21-21.3
  - network flows … Ch. 22.1-22.2
