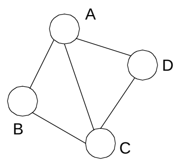
CS211
Graphs Review
David I. Schwartz
Compiled From Spring 2003 CS211 Lecture Notes

1. The Gist

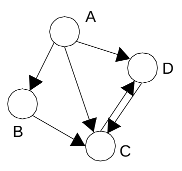
- ***Graphs*** are trees with nodes that may create cycles
- set of ***edges*** (which may have weights and/or directions)
- set of ***verticies*** (or nodes)
- $|V|$ = size of V, $|E|$ = size of E
- generalization of many other data structures!
- example:



•

1.1 Directed Graphs

- also called ***digraphs***
- G = (V,E)
- edges have 1 direction
- write edge as ordered pair (s,d) (source, destination) or s→d
- an edge may have node connect to itself (s==d)
- for 2-way direction, use another edge
- example:



Directed Graph G = (V,E)

Vertices = V = {A,B,C,D}

Edges = E =

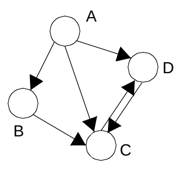
{(A,B),(B,C),(A,D),(A,C),(C,D),(D,C)}

Example: Edges (D,C) and (C,D)

are different!

1.2 More Directed Graphs Terms

- ***adjacency***: for (a,b), b is adjacent to a because there is an edge connecting b to a (reverse is not true, because of directed graph)
- ***out-edges*** of node n: set of edges whose source is n
- ***out-degree*** of node n: number of out-edges of n
- ***in-edges*** of node n: set of edges whose destination is n
- ***in-degree*** of node n: number of in-edges of n



adjacency

out-edge

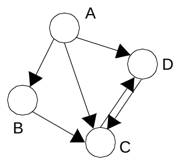
out-degree

in-edge

in-degree

1.3 Continuing Directed Graph Terms

- ***path***: sequence of edges in which destination node of an edge is source node of next edge in sequence; also, set of vertices that satisfy the same property
  ex) edge def: (A,B),(B,C),(C,D)
  ex) node def: A,B,C,D
- ***length of path***: number of edges
- ***cost of path***: sum of weights of edges on path; some references might lump this notion in with length
- ***source of path***: source of first edge on path
- ***destination of path***: destination of last edge on path
- ***reachability***: nodes n is reachable from node m is there is a path from m to n (might have many paths between nodes)
- ***simple path***: a path in which every node is the source and destination of at most two edges on the path (*path does not cross vertex more than once*)



1.4 Cycles

- ***cycle***: a simple path whose source and destination nodes are the same
- length of cycle: length of path (depends on choice of nodes or edges for description)
- loop: path (a,b),(b,a) (edges) or (a,a) (nodes)

2. More Graph Types/Qualities

2.1 Undirected Graphs

- edges have no arrows, so use set for edges: {a,b}
- can go any direction on edge
- nodes cannot form loops ({a,a} becomes just {a})

2.2 Directed Acyclic Graphs

- also called DAGs
- digraph with no cycles
- note: trees are DAGs (but not vice versa)

2.3 Connected Graphs

- a graph with path between every pair of distinct verticies
- disconnected graph includes "lone wolf" nodes (no edges)

2.4 Complete Graphs

- edge between every pair of distinct vertex

2.5 Labeled Graphs

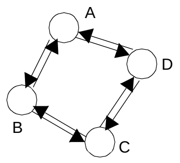
- attach additional info to nodes and/or edges
- ***weights***/***costs***: values on edges (best/worst edges)
  - edge ex) choosing shortest/quickest/best roads to take to get between towns
  - node ex) importance of reaching certain towns ("fun quotient")
- also called ***weighted graphs***

2.6 Trees?

- yes, directed acyclic graphs
- see Tree notes for pretty much the same definitions of vertex and edge

2.7 Sparse and Dense Graphs

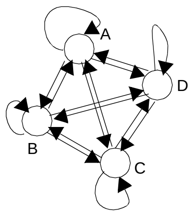
- sparse: not many edges
  - $|E| = O(|V|)$
  - ex) graph with same number of edges emanating from nodes has $|E| = k|V|$, so $|E| = O(|V|)$



$|V| = 4$

$|E| = \left(2\frac{edges}{node}\right)(4nodes) = 8edges$

- dense: many edges
  - $|E|$ essentially on the order of $|V|^2$
  - see pg. 546 DS&A (Def 16.6) for more precision



$|V| = 4$

$|E| = 16$

3. Representations

3.1 Implicit

- rules/model creates a network of nodes/edges
- ex) puzzle moves
  - each move makes a new puzzle
  - treat each state as a node
  - so, rules implicit define a graph
- common for games!

3.2 Explicit

- define all nodes V and edges E ahead of time
- want system to represent edges
- why? it's the "biggest problem":
  - $G = (V,E)$ and each edge e in E is a pair (v1,v2)
  - most edges possible? $|V|$^2
    (form pairs from all nodes)
  - most sets of edges possible? 2^($|V|$^2)
- so, use container to represent edges
  - adjacency matrix
  - adjacency list

3.3 Adjacency Matrix

- ***adjacency matrix***

$$A_{ij} = \begin{cases} w_{ij} & \{v_i, v_j\} \in E \\ 0 & otherwise \end{cases}$$

- terms

  v_i : node i; v_j node j

  $\{v_i, v_j\} \in E$: edge between nodes i (v_i) and j (v_j)
  belongs to set of edges E

  w_{ij} : weight of edge between nodes i and j

- A_{ij} : the matrix (rectangular 2x2 array) as rows (i) and cols (j); coords correspond to nodes i and j

3.4 Adjacency List

- adjacency list: linked list of nodes adjacent to a node
- need $|V|$ lists

3.5 graph types to develop:

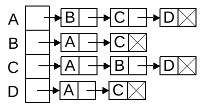
- undirected
- directed
- weighted

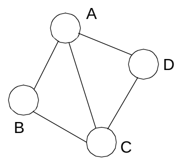
3.6 Undirected

$$A_{ij} = \begin{cases} 1 & \{v_i, v_j\} \in E \\ 0 & otherwise \end{cases}$$

Use array A of lists:
A stores a linked list of nodes
no edge implied by order in list
nodes must be adjacent to A

i \ j	A	B	C	D
A		1	1	1
B	1		1	
C	1	1		1
D	1		1	



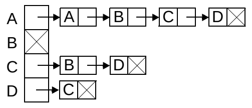


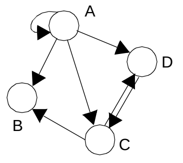
3.7 Directed

$$A_{ij} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & otherwise \end{cases}$$

Use array A of lists:
Ai stores a linked list of nodes
no edge implied by order in list
nodes must be adjacent to Ai

i \ j	A	B	C	D
A	1	1	1	1
B				
C		1		1
D			1	





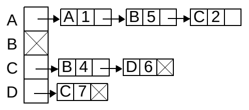
3.8 Weighted

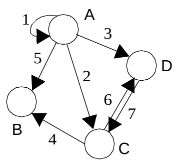
- assuming also weighted
- w_{ij} : cost or weight of edge from node i to node j

$$A_{ij} = \begin{cases} w_{ij} & (v_i, v_j) \in E \\ 0 & otherwise \end{cases}$$

Use array A of lists: include weights
List for i contains j,w for edge (i,j)

i \ j	A	B	C	D
A	1	5	2	3
B				
C		4		6
D			7	





3.9 Choice of AM or AL?

- Adjacency Matrix
  - uses $O(|V^2|)$ space
  - can answer "is there an edge from i to j?" in $O(1)$ time
  - enumerating all nodes adjacent to i: $O(|V|)$ (find all nodes j in row for i)
  - could be sparse because of wasted space (0s)
  - better for dense graphs (lots of edges)!
- Adjacency List
  - uses O(|V|+|E|) space (|V| for i nodes, |E| for j nodes emanating from each i node)
  - can answer question "is there an edge from i to j?" in $O(|E|)$ time
  - enumerating all nodes adjacent to i: $O(1)$ per adjacent node in linked list
  - better for sparse graphs (few edges)!

4. Interesting Problems

4.1 Paths

- find ways to reach/find/collect/organize information from network of nodes
- focus of a lot of research!

4.2 Reachability

- is there a path from a given node to another node?
- ex) find the solved state of N-Puzzle from scrambled state

4.3 Minimal Path

- find the shortest path from a node to another
- find the shortest path from every node to another
- use weights to find min/max distances

4.4 Cycles

- ex) Traveling Salesman problem
- find the smallest length cucle that passes through all nodes
- no one knows if there is an efficient algorithm for this (NP/NP-complete problems)

5. Exercises

- Show all edges and verticies of a 2x2 N-Puzzle.
- Demonstrate a scenario/game/model that forms an implicit graph.
- Demonstrate why we use edges for explicit representations of graphs.

6. Implementation

6.1 Implicit

- can use containers to store node and edge info
- a bit too problem specific, though effective

6.2 Explicit

- Adjacency Matrix - left as exercise
- Adjacency List
  - using linked list to allow for flexible building
  - kind of gives implicit building by allowing for node/ edge creation "on the fly"
- focus on digragh, but could be weighted
  - Sections 3, 4, 5, 6
  - many methods left out – will see for graph problems

7. Verticies

7.1 Fields

- **label**: we like to have names, numbers, …
- **edges**: collection of all emanating edges from the current vertex
- **visited**: need later to tag vertex for searching…
- sometimes includes **cost** (cost to get *here* from somewhere)

7.2 Constructor

- set **label**
- create **edges** adjacency list (AL)

7.3 Methods

- **addEdge**: add to AL
- **equals**: need for path checking
- more?

```
import java.util.*;

public class Vertex {

    private Object label;
    private LinkedList edges; // adjacent edges
    private boolean visited;  // tag

    public Vertex(Object o) {
        label = o;
        edges = new LinkedList();
    }

    public void addEdge(Edge e, int weight) {
        Vertex source = this;
        Vertex dest = e.getDest();
        edges.add(new Edge(source,dest,weight));
    }

    public void addEdge(Edge e) {
        addEdge(e,0);
    }

    public boolean equals(Vertex other) {
        return label.equals( ((Vertex)other).label );
    }

    public String toString() {
        return label.toString();
    }

    public Collection getEdges() { return edges; }

} // Class Vertex
```

8. Edges

8.1 Fields

- source: s->d, the node from which edge emanates
- dest: actually, all you need is this since Vertex keeps track of adjacent edges of source
- weight: could make double (sometimes called cost)

8.2 Constructors

- build edge from s->d
- can default to weight of 0 to handle unweighted graphs

8.3 Methods

- **equals** and **compareTo**:
  - many algorithms want to know shortest path
  - need to compare costs of going in different directions
- **toString**: "**source-weight->dest**"
- more?

```
public class Edge implements Comparable {

    private Vertex source; // s (s->d)
    private Vertex dest;   // d
    private int weight;    // also called cost

    public Edge(Vertex source, Vertex dest, int weight) {
        this.source=source;
        this.dest=dest;
        this.weight=weight;
    }

    public Edge(Vertex source, Vertex dest) {
        this(source,dest,0);
    }

    // getters and setters not shown

    public boolean equals(Object other) {
        Edge e = (Edge) other;
        return weight == e.weight;
    }

    public int compareTo(Object other) {
        Edge e = (Edge) other;
        return (int) (weight-e.weight);
    }

    // Stringify as (d,--w->,s):
    public String toString() {
        return "("+source+"-"+weight+"->"+dest+")";
    }

} // Class Edge
```

9. Directed Graphs

9.1 Fields

- `verticies` dictionary:
  - key-val pairs of (VertexName,Vertex)
  - each Vertex points to its adjacency list!
- `edgeCount`

9.2 Constructors

- set `verticies` to LinkedHashMap
- maintains order of nodes in order created
- nodes *must* be created before edges this way!

9.3 Methods

- use vertex names/labels!
- `addVertex`: put `Vertex` in `Map`: (`name`, `Vertex`)
- `addEdge`: connect s and d nodes (they must exist!)

```
import java.util.*;
public class Digraph {

    private Map verticies; // dictionary of nodes
    private int edgeCount; // number of edges

    public Digraph( ) {
        verticies = new LinkedHashMap();
    }

    // Add vertex to map
    public void addVertex(Object name) {
        verticies.put(name, new Vertex(name));
    }

    // Adds edge (source and dest node must exist!):
    public void addEdge(Object s, Object d, int weight) {

        // Key is NAME of Vertex
        // Val is THE Vertex
        // So, get keys of s and d and use them to
        //      retrieve their vals (their Verticies):
        Vertex source = (Vertex)verticies.get(s);
        Vertex dest = (Vertex)verticies.get(d);

        // Create edge between source and dest:
        s.addEdge(new Edge(source,dest,weight));
        edgeCount++;
    }

    public void addEdge(Object source, Object dest) {
        addEdge(source,dest,0);
    }
```

```
    // Stringify: return edges with
    //            their adjacency lists:
    public String toString() {

        String s = "";

        Iterator it=verticies.keySet().iterator();

        while(it.hasNext()) {

            //  current node label:
            Object key = it.next();

            // current Vertex:
            Vertex val = (Vertex) verticies.get(key);

            // build string for current vertex in Map:
            s += "[" + val + "]" + "-->";
            s += val.getEdges();
            s += "\n";

        }

        return s;

    } // Method toString

} // Class Digraph
```

10. Demonstration

10.1 Code

```
public class TestDigraph {

    public static void main(String[] args) {

        Digraph g = new Digraph();
                g.addVertex("A");
        g.addVertex("B");
        g.addVertex("C");
        g.addEdge("A","B");
        g.addEdge("A","C");
        g.addEdge("B","C");
        System.out.println(g);

    }
}
```

10.2 Output

```
[A]-->[(A-0->B), (A-0->C)]
[B]-->[(B-0->C)]
[C]-->[]
```

11. Exercises

- Demonstrate why we use edges for explicit representations of graphs.
- Develop `Vertex`, `Edge`, `Digraph`, and `TestDigraph` classes for the adjacency matrix approach. You should develop methods to handle I/O in reading in a grid of adjacencies to help build a graph.
- Remove the `source` node field from class `Edge` and modify the remaining classes as necessary. This design is a bit more common than the examples given to you.
- Rewrite `Digraph`'s `addEdge` such that it does not assume that the nodes exist. You may either throw an exception or perhaps create more nodes….
- Graphical graph: This was once a final project long ago…develop a GUI tool that draws a graph that a user creates, either via the GUI or as a translation from the collection that contains the verticies and edges. A rudimentary application would naively draw each vertex according to a pre-determined grid and then draw the edges using the given vertex geometry.

12. Motivation

12.1 Paths

- find ways to reach/find/collect/organize information from network of nodes
- focus of a lot of research!

12.2 Reachability

- is there a path from a given node to another node?
- ex) find the solved state of N-Puzzle from scrambled state

12.3 Minimal Path

- find the shortest path from a node to another
- find the shortest path from every node to another
- use weights to find min/max distances

12.4 Cycles

- ex) Traveling Salesman problem
- find the smallest length cucle that passes through all nodes
- no one knows if there is an efficient algorithm for this (NP/NP-complete problems)

13. Search

- kind of handy that we have explicit graph
- why?
  - verticies created ahead of time
  - stored in hash table
  - so, only need to look up a node…

```
public boolean search(Object o) {
    if (verticies.get(o)==null)
        return false;
    return true;
}
```

14. Traversal

14.1 Traversal

- like search for node, but now search for everything
- visit nodes
- also called walk
- see lists, trees, …

14.2 Why?

- want to find things
- want to way to something
- want to process everything

14.3 Types

- DFS (`Digraph.java`)
- BFS (`Digraph.java`)
- topological sort
- random
- more?

14.4 Applications

- test for cycles
- connectedness

14.5 DFS

- depth-first search
- process:
  - start with origin
  - visits a neighbor
  - visits neighbor of neighbor and so on…
  - stops when can't find unvisited neighbor
  - backs up to previous node and searches for new unvisited neighbor and so on…
- results in visiting all the nodes in a connected graph
- the DFS path never repeats a node
- show/print path as { v1, v2, … vn }:
  - left side (v1) is origin
  - each visited node inserted to right

```
public SeqStructure getDFS(Object origin) {

    resetVerticies();
    SeqStructure toDo = new StackAsList();
    SeqStructure path = new QueueAsList();

    Vertex originVertex = (Vertex)verticies.get(origin);
    originVertex.visit();
    toDo.put(originVertex);
    path.put(originVertex);

    while(!toDo.isEmpty()) {
        Vertex currentVertex = (Vertex)toDo.get();
        Iterator edges = currentVertex.getEdgeIterator();
        boolean found = false;
        Vertex nextVertex = null;
        while (!found && edges.hasNext()) {
            Edge currentEdge = (Edge) edges.next();
            Vertex trialVertex = currentEdge.getDest();
            if(!trialVertex.isVisited()) {
                found = true;
                nextVertex = trialVertex;
            }
        }

        if (nextVertex != null) {
            toDo.put(currentVertex);
            nextVertex.visit();
            toDo.put(nextVertex);
            path.put(nextVertex);
        }
    }
    return path;
}
```

14.6 BFS

- breadth-first search (called ***level-order*** in trees)
- process:
  - visit origin (record this node)
  - visit each of origin's neighbors (record each node in order visited)
  - visit neighbors of each neighbor (record those nodes) and so forth
- show/print path as { v1, v2, … vn }:
  - left side (v1) is origin
  - each visited "level" is inserted to right of previous node
  - so might wish to think as { {level 1}, {level 2}, … }

```
public SeqStructure getBFS(Object origin) {

    resetVerticies();
    SeqStructure toDo = new QueueAsList();
    SeqStructure path = new QueueAsList();

    Vertex originVertex = (Vertex)verticies.get(origin);
    originVertex.visit();
    toDo.put(originVertex);
    path.put(originVertex);

    while(!toDo.isEmpty()) {
        Vertex currentVertex = (Vertex)toDo.get();

        for (Iterator edges =
                               currentVertex.getEdgeIterator();
                               edges.hasNext(); ) {

            Edge currentEdge = (Edge) edges.next();
            Vertex nextVertex = currentEdge.getDest();

            if(!nextVertex.isVisited()) {
                nextVertex.visit();
                toDo.put(nextVertex);
                path.put(nextVertex);
            }
        }

    }

    return path;

}
```

14.7 Example

- see `TestDigraph.java`
- contains more examples

```
    Digraph g = new Digraph();
    g.addVertex("A");
    g.addVertex("B");
    g.addVertex("C");
    g.addVertex("D");
    g.addVertex("E");
    g.addVertex("F");
    g.addVertex("G");
    g.addEdge("A","B");
    g.addEdge("A","C");
    g.addEdge("B","D");
    g.addEdge("B","E");
    g.addEdge("C","F");
    g.addEdge("C","G");

    System.out.println(g.getBFS("A"));
    System.out.println(g.getDFS("A"));

/*
     BFS: [A B C D E F G]
     DFS: [A B D E C F G]
 */
```

37

15. Shortest Path Algorithms

15.1 Assumptions

- edge-weighted graph (unitary or more)
- weight is a cost of using an edge
- graphs may be directed or undirected
- non-negative edge weights!

15.2 Why?

- want to find best/cheapest/least effort between points
- travelling is classic example

15.3 Terms

- ***weighted path length*** = sum of weights on path
- ***unweighted path length*** = sum of paths (weights = 1)

38

15.4 SSSP

- to find shortest path from A to B, need to find *shortest path from A to all other nodes*
  - why? another node might provide a better path
  - see DS&A 16.4.1: basically, knowing all the path lengths might mean you can find a shorter route
- this problem called ***single-source shortest path*** problem

39

16. SSSP for Unweighted Graphs

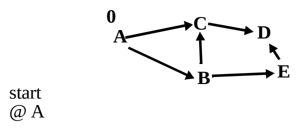
16.1 The Gist

- based on BFS
- all traverse trees, keep track of increasing path lengths to a particular node
- algorithm finds only 1 path if multiple paths are smallest and have same value
- will need to modify classes again
  - need to find all paths to end node
  - need to keep track of length to current node (so can have length values after traversal)
  - so, need to keep track of previous node

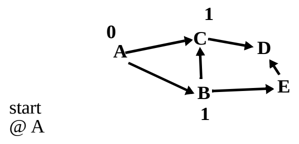
40

16.2 Process for SSSP

- finding shortest path from A to B means counting edges
- smallest number of edges gives shortest path
- since starting at A, start counting @ A:



- try to find final node, so count edges while looking:



- which will get the smallest node cost if check when reach final node and backtrack

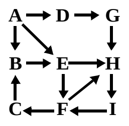
16.3 Path Cost

- for path to node v, distance to v is Dv
- for v $\longrightarrow$ w, Dw = Dv + 1
- helpful convention: default node cost for SSSP: Dw = ∞
- we're not really using the Dw convention, though many implementations do (we have `visited`, which is our way of tagging visited nodes)

16.4 Algorithm (BFS for Destination)

- start with origin
- put origin in Q
- not done, so take first node from Q
- find edges from node
- for each node, if it's not already been visited
  - tag it, set cost, set prev node, put in Q
  - if the node is dest, stop processing!

16.5 Example: Part 1–Build Q



```
origin: A; end: I
get A: Q = [A]
get A: Q = [ ]
    process (AD),(AE),(AB) (nodes D,E,B)
    for each node, put in Q [D,E,B], tag, set cost (1),
    set prev (A), check if dest (no for all)
get D: Q = [E,B]
    process (DG) (node G)
    tag G, put in Q [E,B,G], set cost (G=2),
    set prev (D), check if dest (no)
get E: [B,G]
    process (EH), (EF) (nodes H,F)
    tag nodes, put in Q [B,G,H,F], set costs (H=F=2)
    & prev (E), check if dest (no for all)
get B: [G,H,F]
    nothing new -- no processing
get G: [H,F]
    process (GH)
    nothing new (H already visited)
get H: [
    tag I, put in Q [H,F,I], set cost (3), set prev (H)
    check if dest -- yes! STOP!
```

16.6 Example: Part 2–Build Stack

```
put last vertex (end) into Stack: [I]
set last vertex to prev of last index: H
put last vertex into Stack: [H,I]
set last vertex to prev of last index: E
put last vertex into Stack: [E,H,I]
set last vertex to prev of last index: A
put last vertex into Stack: [A,E,H,I]
no more prev (prev is null)
return Stack, which contains shortest path
```

16.7 Code

```
public SeqStructure unweightedShortestPath(Object origin, Object end) {
      resetVerticies();
      boolean done = false;
      SeqStructure toDo = new QueueAsList();
      SeqStructure path = new StackAsList();

      Vertex originVertex = (Vertex)verticies.get(origin);
      Vertex endVertex = (Vertex)verticies.get(end);
      originVertex.visit();
      toDo.put(originVertex);

      while(!done && !toDo.isEmpty()) {
            Vertex currentVertex = (Vertex)toDo.get();

            for (Iterator edges=currentVertex.getEdgeIterator();
                    !done && edges.hasNext(); ) {

                  Edge currentEdge = (Edge) edges.next();
                  Vertex nextVertex = currentEdge.getDest();

                  if(!nextVertex.isVisited()) {
                        nextVertex.visit();
                        nextVertex.setCost(1+currentVertex.getCost());
                        nextVertex.setPrev(currentVertex);
                        toDo.put(nextVertex);
                  }

                  if (nextVertex.equals(endVertex))
                        done = true;

            } // end for

      } // end while

      path.put(endVertex);
      while(endVertex.hasPrev()) {
            endVertex = endVertex.getPrev();
            path.put(endVertex);
      }
      return path;
}
```

17. Exercises

- Use recursion to rewrite (and simplify) the DFS code. You might need a helper method.
- Write a program that finds all DFS/BFS paths in a graph. Is this problem related to an implicit graph search
- Rewrite the shortest path algorithm such that it uses the convention of “infinite” costs as the check for stopping instead of tagging of nodes.

Try to figure out an algorithm for finding the shortest path when there are edge weights.

18. Overview:

- shortest path algorithm for weighted graph (Dijkstra’s algorithm)
- all pairs source shortest path (Floyd’s algorithm)
- minimum cost spanning trees (Prim’s algorithm, Kruskal’s algorithm)

19. Shortest Path for Weighted Graphs

19.1 Assumptions

- could be directed or undirected
- non-negative weights

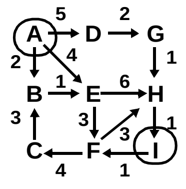
19.2 Dijkstra's Algorithm

- very famous
- example of greedy algorithm
- on-line demo:
  http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/dijkstra/Dijkstra.shtml

19.3 Wordy Gist: Based ON BFS

- BFS: visit the nodes by "levels" or "layers"
  - put new (unvisited) nodes in Q
  - look at each node at each layer
  - process each node and repeat
  - don't re-process already-visited nodes
- New twist!
  - don't treat all unvisited nodes as equals
  - want smallest accumulation of weights
  - so, need to sum weights along the way and maybe pick a different node than what's in front of the Q

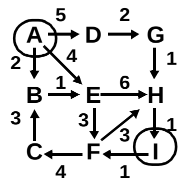
19.4 Physical Gist



Want shortest path from A to I

Imagine graph is weights and strings, in which strings are cut to scaled lengths

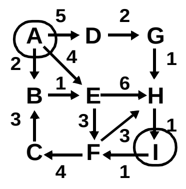
Pick up weights one at a time



Pick up A

String becomes tight *first* at B

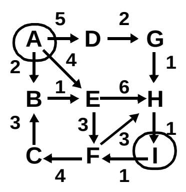
record: A→B



Pick up B

String becomes tight *first* at E

record: A→B→E

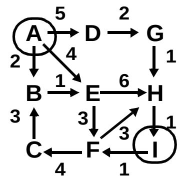


String now becomes tighter at D

Why? After E comes F or H, each of which is longer than D

record: A→D

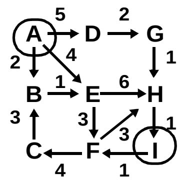
We could have gotten to E via A



Pick up D, followed by F

But, G will be in "next round"

so, record: A→D→G



Eventually:

record: A→D→G→H→I

Forms a tree

19.5 Pseudocode Gist: Version 1

- Longish algorithm that uses cost in organizing priority queue to choose nodes
- a bit expanded on “wordy gist” from before:
  - pick the highest priority node (the smallest dist)
  - tag the node, record previous, update cost:
    PQ element: <node,accumulating cost>
  - repeat until no more PQ or no more unvisted nodes
    (note: tagging happens *after* extract from PQ)
- Visualization:

$$\begin{tikzcd}
& B \arrow[r, "3"] & D \arrow[dr, "4"] & \\
A \arrow[ur, "1"] \arrow[dr, "2"'] & & & F \\
& C \arrow[r, "3"'] & E \arrow[ur, "1"'] &
\end{tikzcd}$$

Current PQ Entry	Current Node	Adjacent Nodes	PQ	Previous Node
<A, 0>	A		[ ]	
		B	[<B,1>]	A
		C	[<B,1>,<C,2>]	A
<B,1>	B		[<C,2>]	
		D	[<C,2>,<D,4>]	A
<C,2>	C			
		E	[<D,4>,<E,5>]	C
<D,4>	D			
		F	[<E,5>,<F,8>]	D
<E,5>	E			
		F	[<F,6>,<F,8>]	E
<F,6>	F			
			[<F,8>]	F

19.6 Code Gist: Version 1

```
// from dijkstra1 in Digraph.java:

resetVerticies();
boolean done = false;
SeqStructure toDo = new Heap(edgeCount); // use min heap!
SeqStructure path = new QueueAsList();// should use stack

Vertex originVertex = (Vertex)verticies.get(origin);
Vertex endVertex = (Vertex)verticies.get(end);
originVertex.setPrev(null);
toDo.put(new MinPQElement(originVertex,0));

while(!done && !toDo.isEmpty()) {
    MinPQElement entry = (MinPQElement) toDo.get();
    Vertex currentVertex = (Vertex) entry.getItem();
    // code not shown


} // end while

path.put(endVertex);
while(endVertex.hasPrev()) {
    endVertex = endVertex.getPrev();
    path.put(endVertex);
}
return path;
```

19.7 Pseudocode Gist: Version 2

- Data:
  - s: start vertex
  - c(i,j): cost from i to j
  - dist(n): distance from s to n (initially ∞)
  - PQ to store neighboring nodes and choose the one with min cost at each “layer”
    (note: PQ size is edgeCount -> max # of adj nodes)
- Algorithm:

```
dist(s) <- 0
while (some vertices are unvisited)
    v <- unmarked vertex with smallest dist
          (get from the PQ)
    tag v
    for each node w adjacent to v
        dist(w) = min(dist(w),dist(v)+c(v,w))
    end for
end while
```

19.8 Code Gist: Version 2

```
public SeqStructure dijkstra3(Object origin,Object end) {

      resetVerticies(Integer.MAX_VALUE);
      SeqStructure toDo = new Heap(edgeCount);
      SeqStructure path = new QueueAsList();

      Vertex originVertex = (Vertex)verticies.get(origin);
      Vertex endVertex = (Vertex)verticies.get(end);

      originVertex.setPrev(null);
      originVertex.setCost(0);
      toDo.put(new MinPQElement(originVertex,0));

      while(!toDo.isEmpty()) {
            MinPQElement entry = (MinPQElement) toDo.get();
            Vertex currentVertex = (Vertex) entry.getItem();
            currentVertex.visit();

            for (Iterator edges=currentVertex.getEdgeIterator();
                                                    edges.hasNext(); ) {

                  Edge currentEdge = (Edge) edges.next();
                  Vertex nextVertex = currentEdge.getDest();
                  int nextCost = currentEdge.getWeight() +
                                              currentVertex.getCost();

                  if (nextVertex.getCost() > nextCost ) {
                        nextVertex.setCost(nextCost);
                        nextVertex.setPrev(currentVertex);
                        toDo.put(
                              new MinPQElement(nextVertex,nextCost));
                  }

            }

      }
      path.put(endVertex);
      while(endVertex.hasPrev()) {
            endVertex = endVertex.getPrev();
            path.put(endVertex);
      }
      return path;
}
```

19.9 Proof Gist

- Induction on iterations of while loop
  - each iteration moves one new node into lifted set
  - do induction on set of nodes ordered in the sequence in which they get put into the lifted set
- Induction:
  - base case: path from origin to self is 0
  - inductive hypothesis: assume that the shortest paths to all nodes currently in the lifted set have been computed correctly
  - inductive hypothesis: the next node that gets lifted is correct
- see Panels 16–19 at http://www.cs.cornell.edu/courses/cs211/2002sp/Lectures/graphs-quad.pdf

19.10 Run-time Analysis for Adjacency List

- dominant operation of method is while loop (processing unvisited nodes)
- time for processing each vertex:
  - each vertex processed once
  - all edges from a vertex might be processed
  - so, for each node, add up time for each edge
  - so, O(|V| + |E|) (see BFS time)
- PQ ops?
  - worst case: each edge has a node to queue and dequeue (see `for` loop and inner `if`)
  - so, PQ has max length of |E|
  - from heap: put is O(log n), get is O(log n)
  - so, adding each edge's contribution gives O(|E| log |E|)
- total: O(|V| + |E| log |E|)

19.11 Adjacency Matrix

- O(|V|^2 + |E| log |E|)

20. All Pairs Shortest Path

20.1 Problem

- given edge weighted graph
- for each pair of verticies find length of shortest path

20.2 One Solution

- run Dijkstra's algorithm |V|+ times
- use each vertex as the origin

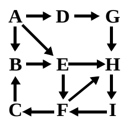
20.3 Floyd's Algorithm

- use adjacency matrix
- see 16.4.2 in DS&A

21. Spanning Trees

21.1 Interesting Thing About Traversals

- BFS, DFS don't repeat -> no cycles
- can backtrack to find a new unvisited node, but won't repeat it
- what does that look like?
- a rooted tree!
- ex) BFS = {A,B,D,E,G,H,F,I,C}



21.2 Spanning Tree

- effectively a subset of a graph:
  - all nodes sames as in G
  - tree edges must be graph edges (but nec all!)
  - connected
  - acyclic
- constructing?
  - pick a starting edge
  - add edges with unvisited dest nodes

21.3 Minimal Spanning Tree

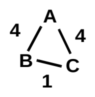
- given: undirected, weighted graph
- weight of spanning tree = sum of tree edge weights
- ***minimum spanning tree***:
  - any spanning tree with smallest weight
  - could have many such trees

21.4 Application

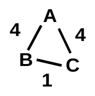
- find a cheap way to connect a bunch of nodes
  - as in something travelling an entire graph
  - plane needs to travel to a set of cities
  - wants cheapest path to take that still hits all cities

21.5 Compare to SSSP

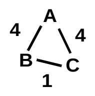
- SSSP: shortest path to a node
  what's cheapest way to get from A to Z using nodes {A,…,Z}
- MST: smallest sum of weights connecting each node
  what's cheapest way to connect all nodes {A,…,Z}?



weighted, undirected graph



SSSP for A→C: {A,C}
Tree: { {A,B}, {A,C} }



MST for graph
Tree: { {A,B}, {B,C} }

21.6 Prim's Algorithm

- modify Dijsktra's Algorithm:
  - put edges in PQ
  - associate edges with length of edge (don't add costs)
  - otherwise, algorithm is the same

21.7 Kruskal's Algorithm

- add edges by increasing order of weights
- not allowed to add edges that form cycles

22. Exercises

- Modify the heap code to use a minimum heap.
- Modify the heap code to provide a sorted string for describing a priority queue.
- Prove by induction that Dijkstra's algorithm is correct.
- Implement Prim's algorithm.