Graph Properties and Types

- studied extensively in mathematics
- examples:
    - maps
    - hypertexts
    - circuits
    - schedules
    - transactions
    - matching
    - networks
    - program structure
- any linked data structure is a representation of a graph, and many familiar algorithms for processing trees and other linked structures are special cases of graph algorithms
- performance characteristics are difficult to analyze because:
    - the cost depends on both nodes and edges
    - accurate models of the range of types of graphs are difficult to develop

Glossary

- definitions:

    - graph: set of vertices and a set of edges connecting distinct pairs of vertices
    - vertex (node): a single element in a graph (vertex for graph, node for representation)
    - edge (arc, link): a single connection between two distinct elements in a graph
    - parallel edges: duplicate edges
    - multigraph: a graph containing parallel edges
    - self-loop: an edge that connects a vertex to itself
    - simple graph: a graph with no self-loops or parallel edges
    - adjacent: two vertices that are connected are adjacent to one another
    - incident: an edge is incident to both vertices that it connects
    - subgraph: a subset of a graph's edges that also constitutes a graph
    - induced subgraph: an identified subset of a graph's vertices and the associated edges
    - graph drawing: placement of a graph's vertices on a plane and graphically depicting the graph in this manner
    - planar graph: a graph that can be drawn in a plane without any edges crossing
    - Euclidean graphs: a graph that specifically relates vertices in a plane with relevant distances

- isomorphic: two graphs are isomorphic if their labels can be interchanged to make the set of edges identical
- path: a sequence of vertices in which successive vertices are adjacent (or the set of edges that defines these connections)
- simple path: a path with distinct edges and vertices
- cycle: a simple path where the initial and final vertices are the same vertex
- cyclic path: a path with the same initial and final vertex (that is not necessarily simple)
- tour: a cyclic path that includes every vertex
- length: the number of edges in a path or a cycle
- disjoint: two paths are disjoint if they have no vertices in common (other than possibly their endpoints)
- vertex disjoint: another name for the general definition of disjoint
- edge disjoint: two paths that have no edges in common (a slightly stronger condition)
- connected graph: a graph with a path from every vertex to every other vertex
- unconnected graph: a graph that consists of connected components that is not connected
- connected components: maximally connect subgraphs
- tree: an acyclic connected graph
- forest: a set of trees
- spanning tree: a subgraph of a connected graph that contains all of that graph's vertices and is a single tree
- spanning forest: a subgraph of a graph that contains all of the graph's vertices and is a forest
- complete graph: a graph with all edges present
- complement: a graph that is the complete graph with all of the edges of the original removed
- union: the graph created by creating a union of both graphs sets of edges
- clique: a complete subgraph
- degree (valency) (of a vertex): the number of edges incident to a vertex, with loops counted twice
- density: the average vertex degree, or 2 * E/V
- dense graph: a graph whose average vertex degree is proportional to V (or E is proportional to V^2)
- sparse graph: a graph whose complement is dense
- bipartite graph: a graph that can be divided into two sets of vertices such that all edges connect a vertex in one set with a vertex in the other
- undirected graph: graphs with no direction applied to edges (i.e. all edges are bidirectional)
- directed graph (digraph): a graph with edges that are one-way or directed, i.e. a pair specifies being able to move from one vertex to the other but not move in reverse

  - directed edges: an edge that specifies an ordering, i.e. the ability to move from one vertex to another but not vice versa
  - source (head): the first vertex in a directed edge
  - destination (tail): the second vertex in a directed edge
  - indegree: a measurement of the number of edges that include a given vertex as the destination
  - outdegree: a measurement of the number of edges that include a given vertex as the source
  - directed cycle: a cycle in a directed graph in which all adjacent vertex pairs appear in the order indicated by the directed edges
  - directed acyclic graph (DAG): a directed graph with no directed cycles (not the same as an acyclic undirected graph)
  - acyclic undirected graph: aka tree
  - underlying undirected graph: a graph defined by the set of edges in a digraph interpreted without a direction
  - weights: a distance or cost associated with an edge
  - weighted graph: a graph that has weighted edges
  - networks: weighted digraphs

- properties:

  - a graph with V vertices has at most V * (V - 1) / 2 edges (V ^ 2 possible pairs of edges includes self-loops and counts twice for each edge)
  - identifying isomorphic graphs is difficult because there V! ways to label vertices
  - a graph G with vertices V is a tree if and only if (NOTE: each of these conditions is sufficient on their own):
    - G has V - 1 edges and no cycles
    - G has V - 1 edges and is connected
    - Exactly one simple path connects each pair of vertices on G
    - G is connected, but removing any edge disconnects it
  - all graphs that have V vertices are subgraphs of the complete graph that has V vertices
  - the total number of different graphs with V vertices is 2 ^ (V * (V- 1) / 2)

Graph ADT

- general graph ADT:

```
struct Edge {
    int v, w;
    Edge(int v = -1, int w = -1) : v(v), w(w) { }
};

class Graph {
private:
    // Implementation-dependent code
public:
    Graph(int, bool); // int -> memory allocation
    ~Graph();
    int V() const;
    int E() const;
    bool directed() const;
    int insert(Edge);
    int remove(Edge);
    bool edge(int, int);
    class adjIterator {
    public:
        adjIterator(const GRAPH &, int);
        int beg();
        int nxt();
        bool end();
    };
};
```

- this is the simplest interface to facilitate basic operations and not a general purpose interface
- a general graph interface needs to account for self-loops and parallel edges
- graph iterator example:

```
template <class Graph>
vector <Edge> edges(Graph &G) {
    int E = 0;
    vector <Edge> a(G.E());
    for (int v = 0; v < G.V(); v++) {
        typename Graph::adjIterator A(G, v);
        for (int w = A.beg(); !A.end(); w = A.nxt())
          if (G.directed() || v < w)
            a[E++] = Edge(v, w);
    }
    return a;
}
```

- it is often the most useful to just consider a graph as a set of its edges
- a client to print graphs:

```
template <class Graph>
void IO<Graph>::show(const Graph &G) {
    for (int s = 0; s < G.V(); s++) {
        cout.width(2); cout << s << ":";
        typename Graph::adjIterator A(G, s);
        for (int t = A.beg(); !A.end(); t = A.nxt()) { cout.width(2); cout << t <<
        cout << endl;
    }
}
```

- generalized operations on graphs:
    - compute the value of some measure of a graph
    - compute some subset of the edges of a graph

    - answer queries about some property of a graph
- graph processing IO interface:

```
template <class Graph>
class IO {
public:
    static void show(const Graph &);
    static void scanEZ(Graph &);
    static void scan(Graph &);
};
```

- usually include preprocessing (generally in the ctor) and public member functions for querying
- static graphs: graphs with a fixed number of vertices and edges
- will not address dynamic graphs which leads to online algorithms aka dynamic algorithms
- graph connectivity interface:

```
template <class Graph>
class CC {
private:
    // implementation-dependent code
public:
    CC(const Graph &);
    int count();
    bool connect(int, int);
};
```

- graph processing client:

```
#include <iostream.h>
#include <stdlib.h>
#include "GRAPH.cc"
#include "IO.cc"
#include "CC.cc"
main(int argc, char *argv[]) {
    int V = atoi(argv[1]);
    GRAPH G(V);
    IO<GRAPH>::scan(G);
    if (V < 20) IO<GRAPH>::show(G);
    cout << G.E() << " edges ";
    CC<GRAPH> Gcc(G);
    cout << Gcc.count() << " components" << endl;
}
```

Adjacency Matrix Representation

- a V x V matrix of boolean values representing a graph with 0 in a row x column location if there is no edge between vertices and 1 if there is an edge

- adjacency matrices are better suited to dense graphs
- adjacency matrix graph representation:

```
class DenseGRAPH
{ int Vcnt, Ecnt; bool digraph;
  vector <vector <bool> > adj;
public:
  DenseGRAPH(int V, bool digraph = false) :
    adj(V), Vcnt(V), Ecnt(0), digraph(digraph)
    {
      for (int i = 0; i < V; i++)
        adj[i].assign(V, false);
    }
  int V() const { return Vcnt; }
  int E() const { return Ecnt; }
  bool directed() const { return digraph; }
  void insert(Edge e)
    { int v = e.v, w = e.w;
      if (adj[v][w] == false) Ecnt++;
      adj[v][w] = true;
      if (!digraph) adj[w][v] = true;
    }
  void remove(Edge e)
    { int v = e.v, w = e.w;
      if (adj[v][w] == true) Ecnt--;
      adj[v][w] = false;
      if (!digraph) adj[w][v] = false;
    }
  bool edge(int v, int w) const
    { return adj[v][w]; }
  class adjIterator;
  friend class adjIterator;
};
```

- processing all vertices adjacent to a given vertex requires time proportional to V
- adjacency matrix iterator:

```
class DenseGRAPH::adjIterator
{ const DenseGRAPH &G;
  int i, v;
public:
  adjIterator(const DenseGRAPH &G, int v) :
    G(G), v(v), i(-1) { }
  int beg()
    { i = -1; return nxt(); }
  int nxt()
    {
      for (i++; i < G.V(); i++)
        if (G.adj[v][i] == true) return i;
      return -1;
    }
  bool end()
```

```
    { return i >= G.V(); }
};
```

- saving space:
  - undirected graphs are symmetric, i.e. `a[v][w]` always equals `a[w][v]` so only half of the matrix needs to be stored
  - can use n X n bitsets (vector is suggested but this data type is problematic in C++)
- associating vertex names with integers from 0 - n:
  - hashing
  - adding an index field to each node in an existence table TST
- two seemingly different graphs can potentially be equivalent through vertex renaming
- do not use adjacency matrices for large sparse graphs (due to the use of V^2 space)

Adjacency List Representation

- preferred standard representation
- for undirected graphs, add an edge by adding vertex v to w's adjacency list and w to v's adjacency list
- be sure to include destructors, copy constructors (and move constructors, assignment operators) for various implementations
- includes array (or vector) of linked lists

```
// Program 17.9 here Sparse_multigraph
// Program 17.10 here Sparse_multigraph_iterator
```

- basic implementations do not remove parallel edges on edge insertion (constant time vs. time proportional to V to remove parallel edges)
- this approach is not suitable for large adjacency lists
- can associate non-integer vertex names with integers but cannot determine isomorphism because of the difficult of the graph isomorphism problem
- numerous representations of the same graph even for a set numbering
- primary advantage is the use of space E + V (as opposed to V^2)

Variations, Extensions, and Costs

- methods for improvement:
  - adjacency matrices and adjacency lists can be extended easily
  - additional features with advanced data structures
  - graph processing with task specific classes
- examples:
  - digraphs may need a method for representing edges coming into a vertex
  - weighted graphs and networks use structures containing additional Edge information
  - can use vertex indexed vectors to associate additional information with vertices

    - often use special classes for graph processing with additional data structures to hold vertex information

```
// Program 17.11 Degree class
```

- "fat" interfaces can have serious drawbacks for graph processing
- an alternative to separate graph processing classes is inheritance (can be an exercise)

```
// Table 17.1 Worst-case cost of graph processing algorithms
```

- find edge and remove edge are often used and so it is sometimes beneficial to use auxiliary symbol tables to improve search, etc (std::map, std::set, etc)
- must use pointers for auxiliary symbol tables, also need doubly linked lists for efficiency
- vertex removal is also expensive because any edges referencing that vertex must also be removed
- these operations are omitted from the text because they all relate to previous exercises, algorithms, etc
- static graphs can replace the linked lists with vectors which can improve worst case run times
- graph processing algorithms are not linear in time if an adjacency matrix is used to represent a sparse graph or if an adjacency list is used to represent an extremely sparse graph (ignored due to standard assumptions in text)

Graph Generators

- can generate graphs from user input or randomized edges

```
// Program 17.12 Random graph generator
```

- generate pairs from 0 to V - 1 (will likely contain self loops and parallel edges)
- removing parallel edges will generally lead to a more dense graph
- can also randomly generate the number of vertices

```
// Program 17.13 Random graph generator
```

- have a well-studied mathematical/probabilistic background
- can choose a probability, p, such that p = 2 * E / V (V - 1)
- k-neighbor graph:
    - randomly pick a vertex, v, then randomly pick the second within a constant k of v (wrapping from V - 1 to 0)

- Euclidean neighbor graph:
  - generate V points in a plane with random coordinates between 0 and 1 and then generates edges for all points within a distance d of each other
  - if a sparse graph is generated, the graph is not likely to be connected
- transaction graph:
  - used to model connections between elements with id's (i.e. telephone calls, etc)
- can use a symbol table to build a graph by equating a symbol with its index

```
// Program 17.14 Building a graph from pairs of symbols
```

- function call graph:
  - a computer program can be represented as a graph with functions as vertices and edges representing when a function is called from within another function
  - can be generated manually or instrumented within compilers
- can use a TST to index string keys for a graph

```
// Program 17.15 Symbol indexing for vertex names
```

- often need to determine a way to map symbolic names to vertex indices to represent a problem as a graph
- degrees-of-separation graph:
  - collection of subsets from V, item is given a degree of separation from a given vertex, v, indicating the number of incident steps it takes to arrive at that item from v
- interval graph:
  - intervals are defined on a number line, assigned as vertices, and edges are defined if the intervals intersect
- de Brujin graph:
  - digraph with V as a power of 2, with one vertex for each non-negative integer less than V, and edges from each vertex i to 2i and to (2i + 1) % lg(V)

Simple, Euler, and Hamiltonian Paths

- simple DFS path search

```
// Program 17.16 Simple path search
```

- some simple path problems just want to know if a path exists, others want to find a specific path

- can print a path using DFS by switching the order of the inputs and printing the edge after the recursive call (switch order because otherwise the printed order would print the reverse path)

- can use this same format to use a client-defined visit function

- properties:
    - we can find a path connection two given vertices in a graph in linear time
- worst case DFS for adjacency lists checks all of the edges twice (once in each direction)
- linear in graphs means within a constant factor of V + E
- cannot refer to V^2 as linear and adjacency matrix algorithms that check all graph edges are always V^2
- paths may be found before examining all edges, thus reducing running time
- Hamilton path -> a path between two vertices containing each vertex exactly once
- Hamilton tour -> a Hamilton path that returns to the original vertex

```
// Program 17.17 Hamilton path
```

- properties:
    - a recursive search for a Hamilton tour could take exponential time
    - if there is not a simple path or a Hamilton path from t to w, then there is not one for v to w that goes through t (where v is connected to t)
- the result is that every single path in a graph may need to be checked, resulting in a (V - 1)! number of recursive calls or roughly (V / e) ^ V
- Euler path -> a path between two vertices that uses each edge exactly once
- Euler tour -> a cyclic path that uses each edge exactly once
- generally considered the beginning of graph theory
- Bridges of Konigsberg
- properties:
    - a graph has an Euler tour if and only if it is connected and all its vertices are of even degree
    - corollary - a graph has an Euler path if and only if it is connected and exactly two of its vertices are of odd degree

```
Program 17.18 Euler path existence
```

- with induction, to have an Euler tour, need to leave the start vertex on one edge and return on another, etc

- to check if an Euler path/tour exists, can check the vertex degrees with a matrix in V ^ 2 time, time proportional to E with an edge set, and time proportional to V with a vertex indexed vector representation

- to find an Euler path, run into the same factorial time issue as above, but can reduce running time by testing edges before following them

- properties:

  - an Euler tour in a graph (that has an Euler tour) can be found in linear time

- the process follows a cycle by pushing the vertices onto a stack and backtracks on failure to check additional side paths

```
Program 17.19 Linear-time Euler path
```

- few graphs have Euler tours and, as such, these algorithms are not broadly applicable
- DFS is the most common generalized graph search method

Graph Processing Problems

- often useful to first determine how difficult a problem is to solve

- problem types:

  - simple connectivity - checks if there is a path connecting every pair of vertices (generally easy)
  - strong connectivity in digraphs - is there a directed path connecting every pair of vertices (generally easy)
  - transitive closure - the set of vertices that can be reached by following directed edges from each vertex in a digraph (generally easy)
  - minimum spanning tree - in a weighted graph, the minimum-weight set of edges that connects all vertices (classical problem)
  - single-source shortest paths - shortest paths connecting a vertex with each other vertex in a weighted digraph (network) (difficult if edge weights can be negative)
  - planarity - draw a graph without any edges (lines) intersecting -> Kuratowski, Tarjan
  - matching - largest subset of a graph's edges with no two connected to the same vertex (also bipartite matching)
  - even cycles in digraphs - cycle of even length in a digraph (difficult to prove)
  - assignment - bipartite weighted matching -> perfect matching of minimum weight in a bipartite graph
  - general connectivity - minimum number of edges that can be removed that will separate a graph into two disjoint parts (difficult, related to network flow)
  - mail carrier - a tour with a minimal number of edges that uses every vertex at least once (more difficult than Euler tour and less difficult that Hamilton tour)

- intractable problems:
  - longest path - longest simple path connecting two vertices (NP-hard)
  - colorability - find a method of assigning k colors to each of the vertices in a graph such that no edge connects two vertices of the same color (easy for k of 2, NP-hard for k of 3)
  - independent set - largest subset of the vertices such that not two are connected (NP-hard)
  - clique - size of the largest clique (complete subgraph)
  - graph isomorphism - make two graphs identical by renaming vertices
- see table 17.2 for more info

Graph Search

- almost all graph problems follow the abstract model set up by graph search

Exploring a Maze

- maze = graph, passage = edge, intersection = vertex
- Tremaux exploration
  - setup - lights (off) in each intersection and doors (closed) at the end of each passage
  - goal - turn on all lights and open all doors
  - process:
    a. if no closed doors -> 3. else open any closed door
    b. if the light at the end of the passage is lit, try another door else (if dark) follow passage and unroll string
    c. if all doors open if at start? then stop else return to previous intersection (re-rolling string) -> 1.
- properties:
  - Tremaux maze exploration lights all lights and opens all doors and returns to maze start

Depth-first Search

- recursive DFS:
  - mark a vertex as visited and recursively visit all adjacent vertices

```
Program 18.1 Depth–first search of a connected component
```

- NOTE: generates a tree-like call stack, visits each edge twice, once as v-w, once as w-v

- difference with Tremaux exploration:
    - results in a slightly different ordering
    - moving on edge v - w, edges w - * are not examined (i.e. w - v) which is ignored when reached because it is marked
    - for a direct correspondence, close doors on the way into a passage and open doors on the way out of a passage
- adjacency lists and adjacency matrices result in a different ordering of visitation i.e. order in lists vs. numerical order
- NOTE: DFS and the following sections use vector ord (order) to mark the order in which the vertex was visited

Graph Search ADT Functions

- following graph edges from vertex to vertex only leads to visiting all vertices in a connected component

```
// Program 18.2 Graph search
```

- general search process (using an ADT):
    - find an unmarked vertex
    - visit and mark as visited all the vertices in the connected component that contains the start vertex
    - the method for finding the next vertex in the next connected component is not strictly specified but generally is found by increasing scan
    - pass an edge rather than the next vertex to know the start point

```
// Program 18.3 Derived class for depth-first search
```

- properties:
    - a graph search function checks each edge and marks each vertex in a graph if and only if the search function that it uses marks each vertex and checks each edge in the connected component that contains the start vertex
    - DFS of a graph represented with an adjacency matrix requires time proportional to V^2
    - DFS of a graph represented with adjacency lists requires time proportional to V + E
    - general DFS-like graph search functions, with time f(V, E), take time E + V * f(V, E)
- use the convention of initializing each index of the visited vertex vector to -1 and positive (1) when visited

- DFS creates a tree of the nodes that are visited in the order they are visited

Properties of DFS Forests

- trees used to represent recursive DFS function calls give insight into DFS
- representations:
    - external nodes as moments when a node was skipped as marked
        - tree represents graph
        - preorder traversal of the tree == DFS traversal of graph (and Tremaux maze traversal)
- NOTE: uses a vector, st, to record parent links
- edge classification:
    - edges representing a recursive call (tree edges)
    - edges connection a vertex to an ancestor not its parent (back edges)
    - links for v to w are classified as:
        - a tree link if w is unmarked
        - a parent link if `st[w]` is v
        - a back link if `ord[w] < ord[v]`
        - a down link if `ord[w] > ord[v]`
- a DFS forest can be used to represent the dynamics of DFS for a given graph
- connected components correspond to different trees in the forest, and the st vector contains a representation of these relationship (consider UF graph representation)
- most resulting considerations are trivial for small graphs but become important for large graphs:
    - recursive depth
    - paths
    - number of adjacent vertices to a given vertex

DFS Algorithms

- cycle detection:
    - a graph is acyclic if and only if no back or down edges are encountered in DFS
    - can exit early - will find a cycle or finish the search without finding one before examining V edges (time proportional to V for list repr, V^2 for matrix repr)
- simple path:
    - use DFS with a visited/marked vector (17.7)
    - linear time

- simple connectivity:
    - connected if recursive DFS is only called once in graph search
    - number of connected components == the number of times recursive DFS function is called in graph search
    - linear time

```
// Program 18.4 Graph connectivity
```

- two-way Euler tour:
    - use Tremaux exploration but go back and forth each back link and ignore down links (or ignore back links and go back and forth down links)

```
// Program 18.5 Two-way Euler tour
```

- spanning forest:
    - find a set of V - 1 edges that connect the vertices of a connected graph for C connected components, find a spanning forest with V - C edges
    - see 18.3
- vertex search:
    - how many vertices are in the same connected component as a given vertex
    - just run DFS from the start vertex and count the number of marked vertices
- two-colorability (bipartiteness) odd cycle
    - assign one of two colors to each vertex such that no edge connects two vertices of the same color
    - equivalent to checking if a graph has a cycle of odd length (odd length cycle is not two colorable)
    - alternate colors while moving down a DFS tree and check for inconsistencies on the way back up

```
// Program 18.6 Two-colorability (bipartiteness)
```

!!!NOTE - revisit this section to fully understand the tree labeling/generation

Separability and Biconnectivity

- general algorithms look at graph connectivity if an edge is removed and if a vertex if removed
- definitions:
    - bridge: edge in a connected graph, that, if removed, would result in two disjoint subgraphs
    - edge-connected: a graph with no bridges
    - edge-separable: not edge connected

- separation edges: bridges
- edge-connected/bridge-connected components: maximal subgraphs with no bridges
- articulation point: vertex in a connected graph, that, if removed, would result in
- separation vertex or cut vertex: articulation point
- biconnected: every pair of vertices is connected by two disjoint paths
- k-connected: at least k vertex disjoint paths connecting every pair of vertices in a graph
- vertex connectivity: minimum number of vertices that need to be remove to separate it into two pieces
- k-edge connected: at least k edge-disjoint paths connecting every pair of vertices in the graph
- edge connectivity: mminimum number of edges that need to be removed to separate it into two pieces

- properties:
    - a tree edge in a DFS tree is a back edge if and only if there are no back edges that connect a descendant of the forward vertex with an ancestor of that vertex
    - a graph's bridges can be found in linear time
    - a graph is biconnected if an only if it has no articulation points
    - a graph's articulation points and biconnected components can be found in linear time

```
// Program 18.7 Edge connectivity
```

- vertex removal implies removal of all incident edges
- related problems:
    - st-connectivity: minimum number of edges or vertices that will separate two vertices, s and t, in a graph
    - general connectivity: k-connectivity, k-edge-connectivity, edge connectivity, vertex connectivity

Breadth-first Search

- breadth-first search yields shortest paths
- swap DFS's stack with a queue
- general steps:
    - start by placing the start vertex on the queue
    - take edges from the queue until finding one that points to an unvisited vertex
    - visit vertex, enqueue all of that vertex's adjacent vertices
- properties:
    - during BFS, vertices enter and leave the queue in the order of their distance from the start vertex
    - BFS visits in V^2 for matrix repr and V + E for list repr

```
// Program 18.8 Breadth-first search
// Program 18.9 Improved BFS
```

- recursive calls of BFS produces a spanning forest that provides specific information about the graph
- can use a global vertex marking vector to avoid visiting duplicate edge vertices
- keep an st vector to record parent links, and an order vector for order visited
- shortest path:
  - use a parent link vector and stop BFS at destination vertex
  - the path from w to v through the parent link vector is the shortest path
  - to get v to w search from w to v or reverse the order using a stack
- single source shortest paths:
  - finds the shortest path from v to all other vertices
  - run BFS to completion and the parent link vector will contain the shortest path from each vertex back to the start vertex
- all-pairs shortest paths
  - run and store the result of single source shortest path for each vertex
  - requires time proportional to V * E and space proportional to V^2

Generalized Graph Search

- both DFS and BFS are examples of generalized graph search algorithm
- fringe - the structure used to store vertices to search
- use an empty tree to store paths
- general steps:
  - add start vertex to fringe
  - move an edge from the fringe to the tree
  - visit the edge if unvisited
  - put all edges on fringe for given vertex that lead to unvisited vertices
  - continue
- fringe -> queue == BFS, fringe -> stack == DFS
- graph search coloring:
  - white -> unvisited
  - black -> in tree/visited
  - gray -> in fringe/yet to be visited

- properties:

  - generalized graph search visits all vertices and edges in V^2 for matrix repr and V + E for list repr plus (in the worst case) the time required for V insert, V remove, and E update operations in a generalized queue of size V

```
// Program 18.10 Generalized graph search
```

- alternatives to BFS and DFS:
  - randomized queues: randomly remove the next vertex from the fringe so each vertex is equally likely to be the next inserted in the tree
  - use a priority queue for the fringe (important for many algorithms) and use highest priority as the next inserted into the tree

```
// Program 18.11 Random queue implementation
```

- can generalize graph search further by using a forest as output rather than a tree

Analysis of Graph Algorithms

- TODO: more work here

Digraphs and DAGs

- edge ordering becomes significant, i.e. edges have a direction

Glossary and Rules of the Game

- definitions:

  - digraph: vertices with directed edges that connected ordered pairs of vertices with no duplicate edges (directed edges go from first vertex to second vertex)
  - directed path: a list of vertices with a directed edge connecting each vertex
  - reachable: a vertex t is reachable from s if there is a directed path from s to t
  - map: a digraph where self loops are allowed
  - indegree: number of directed edges that lead to a vertex
  - outdegree: number of directed edges leading out of a vertex
  - directed acyclic digraph: a directed graph with no cycles
  - strongly connected digraph: a digraph where every vertex is reachable from every other vertex
  - strongly connected or mutually reachable: a pair of vertices in a digraph that have a directed path from one to the other and vice versa

    - kernel DAG: see the second property below

- reversing a digraph:

    - take an input digraph and an empty output digraph and insert the reversed edge to the output digraph
    - for the matrix repr -> can interchange rows and columns or just swap s and t when accessing an element

- properties:

    - a digraph that is not strongly connected is a set of strongly connected subgraphs with a directed edge between them
    - given a digraph, D, and a digraph K(D) defined such that each vertex corresponds to a strongly connected subgraph of D and each edge in K(D) corresponding to an edge that connects those components in D, then K(D) is a DAG (aka the kernel DAG of D)

- NOTE:

    - connectivity -> used for undirected graphs
    - reachability -> generally reserved for use with digraphs
    - strong connectivity -> not a special property for connected vertices in an undirected graph

Anatomy of DFS in Digraphs

- while the general outline is the same, DFS in digraphs is more complicated
- preorder and postorder become more important here
- same edge labeling as above for DFS trees:
    - tree edges: edges representing a recursive call
    - back edges: edges from a vertex to an ancestor
    - down edges: edges from a vertex to a descendant
    - cross edges: edges between a vertex and another vertex that is neither an ancestor nor a descendant

```
// Program 19.2 DFS of a digraph
```

- directed cycle detection:

    - can be achieved in a digraph with DFS by check for back edges

- single source reachability:

    - the set (and count) of reachable vertices from a start give start vertex

- properties:

    - in a DFS forest for a digraph, an edge:
        - to a visited node is a back edge if it leads to a node with a higher postorder number
        - is a cross edge if it leads to a node with a lower preorder number
        - is a down edge if it leads to a node with a higher preorder number
    - a digraph is a DAG if and only if we encounter no back edges when we use DFS to examine every edge
    - the single-source reachability problem can be solved for a vertex s in a digraph in time proportional to the number of edges in the subgraph induced by the reachable vertices

Reachability and Transitive Closure

- TODO: return to this section to take more notes
- definitions:
    - transitive closure: for a digraph, the corresponding digraph with an edge from s to t if and only if there is a directed path from s to t in the original graph
    - boolean matrix: matrix with binary values

```
// dot product of matrices
for (s = 0; s < num_vertices; s++) {
    for (t = 0; t < num_vertices; t++) {
        for (i = 0; c[s][t] = 0; i < num_vertices; i++) {
            // classic version: c[s][t[ += a[s][i] * b[i][t]
            // c++ version: if (a[s][i] && b[i][t]) c[s][t] = 1;
        }
    }
}

// Warshall's algorithm
for (i = 0; i < num_vertices; i++) {
    for (s = 0; s < num_vertices; s++) {
        for (t = 0; t < num_vertices; t++) {
            if (a[s][i] && a[i][t]) a[s][t] = 1;
        }
    }
}
```

- properties:
    - we can compute the transitive closure of a digraph by constructing the by constructing the the graph's adjacency matrix A, adding self-loops for every vertex, and computing A ^ V
    - Warshall's algorithm can compute the transitive closure of a digraph in time proportional to V ^ 3
    - constant time reachability testing (abstract transitive closure) for a digraph using space proportional to V^2 and time proportional to V^3 for preprocessing

    - we can use any transitive-closure algorithm to compute the product of two boolean matrices with at most a constan-factor difference in running time
    - with DFS, we can support constant query time for the abstract transitive closure of a digraph, with space proportional to V^2 and time proportional to V (E + V) for preprocessing (computing the transitive closure)

```
// Program 19.3 Warshall's algorithm
```

```
// Program 19.4 DFS–based transitive closure
```

Equivalence Relations and Partial Orders

- set theory has a relationship to abstract transitive closure algorithms
- definitions:
    - relation: given a set, a set of ordered pairs of the objects in the set
    - symmetric: a relation R if sRt means tRs
    - reflexive: for a relation sRs
    - transitive: sRt and tRu implies sRu
    - partial order: a transitive relation that is also irreflexive (also asymmetric and acyclic)
    - total order: partial order where sTt or tTs for all s != t (check this definition)
- partial orders are related to:
    - subsets
    - paths in DAGs
- list of corollaries between set theory and graphs:
    - relations <-> digraphs
    - symmetric relations <-> undirected graphs
    - transitive relations <-> paths in graphs
    - equivalence relations <-> paths in undirected graphs
    - partial orders <-> paths in DAGs

DAGs

- prototypical DAG problems:
    - scheduling, tasks, constraints
- definitions:
    - scheduling: determine an ordering for a set of tasks with constraints in the form of a partial ordering

    - binary DAG: DAG with two edges leaving each node, identified as the left edge and the right edge, either potentially null
- scheduling is topological sorting
- DAGs are part tree, part graph
- a DAG can be recursively traversed in the same way as a tree
- the distinction between a binary DAG and a binary tree is that a nodes in a binary DAG can have more than one path to it
- binary DAGs can sometimes lead to more compact representations of a binary tree
- trie keys can be viewed as a correspondence to rows in a truth table for a boolean function for which the function is true -> binary DAG can be viewed as an economical circuit that computes the function

```
// Program 19.5 Representing a binary tree with a binary DAG
```

Topological Sorting

- TODO: take more notes here
- source: a vertex with only outgoing edges
- sink: a vertex with only ingoing edges
- two flavors:
    - relabel: given a DAG, relabel its vertices such that every directed edge points from a lower numbered vertex to a higher numbered one
    - rearrange: given a DAG, rearrange its vertices on a horizontal line such that all the directed edges point from left to right
- properties:
    - postorder numbering in DFS yields a reverse topological sort for any DAG
    - every DAG has at least one source and at least one sink

```
// Program 19.6 Reverse topological sort
```

- can reverse the DAG before sorting
- can push the index onto a stack

```
// Program 19.7 Topological sort
```

- NOTE: review pigeonhole principal
- a topological sort algorithm follows from the second property above
- NOTE: review the explanation here

```
Program 19.8 Source-queue based topological sort
```

Reachability in DAGs

- any method for topological sorting can serve as the basis for transitive closure in DAGs by:
  - iterate through vertices in reverse topological order
  - compute the reachability vector for each vertex (the corresponding row in the transitive closure matrix)
- DFS can improve topological sort for some DAGs because there are no back edges and both cross edges and down edges correspond to nodes for which DFS has completed
- in some ways corresponds to using dynamic programming to compute the transitive closure

```
// Program 19.9 Transitive closure of a DAG
```

- properties:
  - using dynamic programming and DFS, constant query time for the abstract transitive closure of a DAG can be achieved (with space proportional to V^2 and time proportional to V^2 + VX for preprocessing, where X is the number of cross edges in the DFS forest)
- the problem of finding an optimal algorithm for transitive closure of a DAG (time proportional to V^2) is still unsolved, with best known worst case bound of V * E

Strong Components in Digraphs

- undirected graphs and DAGs are simpler than general digraphs because there is a structural symmetry that characterizes the reachability:
  - undirected graphs -> s - t implies t - s
  - DAGs - s -> t means not t -> s
  - general digraphs - s -> t give no information about t -> s
- strong connectivity means that if t is reachable from s, then s is reachable from t
- goal:
  - assign numbers to each vertex from 0 - n where n is one less than the total number of strong components in the digraph (using a vertex indexed vector)
  - use these numbers to test whether two vertices are in the same strong component
- brute force:
  - using a transitive closure ADT, check reachability for each vertex in both directions
  - define an undirected graph with an edge for each such mutually reachable pair
- the history of better solutions than the brute force solution is instructive and should be reviewed
- Kosaraju's algorithm:
  - run DFS on general digraph's reverse, computing the permutation of the verties defined by the postorder numbering (topological sort if the digraph is a DAG)
  - run DFS again on the graph to find the next vertex to search and use the unvisited vertex with the highest postorder number

    - when the unvisited vertices are checked according to topological sort in this way, the tress in the DFS forest define the strong components (as trees in a DFS forest define the strong components in undirected graphs)
    - two vertices are in the same strong components if and only if they are in the same tree in the DFS forest

```
// Program 19.10 Strong components (Kosaraju's algorithm)
```

- Tarjan's algorithm:
    - consider vertices in reverse topological order so that when the end of a recursive call is reached, there are no more vertices in the same strong component
    - back links in a tree provide a path from one vertex

```
// Program 19.10 Strong components (Trajans's algorithm)
```

- Gabow's algorithm:
    - Follows Tarjan's algorithm but uses a second stack rather than a vertex indexed vector of preorder numbers to decide when to pop all the vertices in each strong component from the main stack

```
// Program 19.10 Strong components (Gabow's algorithm)
```

- properties:
    - Kosaraju's method finds the strong components of a graph in linear time and space
    - Tarjan's algorithm finds the strong components of a digraph in linear time
    - Gabow's algorithm finds the strong components of a digraph in linear time
- Tarjan's and Gabow's are preferable to Kosaraju's because they only require one pass through the graph and they do not require computation of the reverse for sparse graphs

Transitive Closure Revisited

- using strong components and reachability, a transitive closure algorithm can be developed that offers the same worst case but better general case performance over the DFS-based solution
- process:
    - preprocess:
        - find strong components
        - build the kernel DAG
        - compute the transitive closure of the kernel DAG
    - the information for reachability is then readily available

- properties:
    - given two vertices s and t in a digraph, D, let sc(s) and sc(t) be their corresponding vertices in D's kernel DAG K. Then, t is reachable from s in D if and only if sc(t) is reachable from sc(s) in K
    - can support constant query time for the abstract transitive closure of a digraph with space proportional to V + V^2 and time proportional to E + V^2 + vx for preprocessing (computing the transitive closure) where v is the number of vertices in the kernel DAG and x is the number of cross edges in its DFS forest
    - can support constant query time for the abstract transitive closure of any digraph whose kernel DAG has less the V^(1/3) vertices with space propertional to V and time proportional to E + V for preprocessing

```
Program 19.13 Strong-component-based transitive closure
```

- the primary limiting factor for the applicability of this method is the size of the kernel DAG, i.e. the more similar the digraph is to a DAG (the larger its kernel DAG) the more difficult it is to comput its transitive closure

Perspective

- topics covered:
    - topological sorting
    - transitive closure
    - shortest paths
    - cycles
    - strong components

```
                problem                 cost                algorithm
Digraphs
                cycle detect            E                   DFS
                transitive closure      V * (E + V)         DFS from each vertex
                ss shortest paths       E                   DFS
                all shortest paths      V * (E + V)         DFS from each vertex
                strong components       E                   Kosaraju, Tarjan, or G
                transitive closure      E + V(V + X)        kernel DAG

DAGs
                acyclic verify          E                   DFS or source queue
                topological sort        E                   DFS or source queue
                transitive closure      V * (V + E)         DFS
                transitive closure      V * (V + X)         DFS/dynamic programmin
```

- additional algorithms to consider:
    - dominators: given a DAG with all vertices reachable form r, a vertex s dominates a vertext if every path from r to t contains s

    - transitive reduction: given a digraph, find a digraph that has the same transitive closure and the samlest number of edges among all such digraphs
    - directed Euler path: given a digraph, is ther a directed path connecting two vertices that uses each edge in the digraph exactly once
    - directed mail carrier: a directed tour with a minimal number of edges that uses every edge in the graph at least once and can use edges multiple times
    - directed Hamilton path: find the longest simple directed path in a digraph (NP-hard but simple for DAGs)
    - uniconnected subgraph: uniconnected - at most one directed path between any pair of vertices for a digraph and an integer k, determine whether there is a uniconnected subgraph with at least k edges (NP-hard)
    - feedback vertex set: is there a subset of at most k vertices that contains at least one vertex from every direct cycle (NP-hard)
    - even cycle: find a cycle of even length in digraph (not intractable but not adequately solved)

Minimum Spanning Trees

- many graph problems require associating weights with edges (distance, time, cost, etc)
- important to not think of weights as distances by default
- definitions:
    - minimum spanning tree: a spanning tree whose total weight is no larger than the weight of any other spanning tree
- if edges have equal weights, the MST may not be unique
- the term minimum may be misleading -> it must be specified that it refers to weight

Representations

- in the matrix repr:
    - edges contain weights instead of boolean values
- in the list repr:
    - edges in lists contain an extra weight field

```
// Program 20.1 ADT interface for graphs with weighted edges
```

```
// program 20.2 Example of a graph-processing client function
```

- often a better alternative to define edge structures and manipulate pointers to edges

```
// Program 20.3 Weighted-graph class (adjacency matrix)
// Program 20.4 Iterator class for adjacency-matrix representation
```

- test for the existence of an edge by checking whether the pointer in the row/column is null
- the flexibility provided by the pointers is often worth the space cost of constructing Edge structures

```
// Program 20.5 Weighted graph class (adjacency lists)
```

- do not explicitly test for parallel edges in either implementation
- MST representations:
    - a graph (MST is a sparse graph)
    - a linked list of edges (in no particular order) -> most straightforward
    - a vector of pointers to edges
    - a vertex-indexed vector with parent links (most compact) -> use two, one with parent, other with the weight
- can convert from any one representation to any other in linear time
- default to using an MST ADT class with a private vector of pointers to edges

Underlying Principles of MST Algorithms

- definitions:
    - cut: a partition of the vertices into two disjoint sets
    - crossing edge: an edge that connects a vertex in one set with a vertex in the other
- properties:
    - (cut property) given any cut in a graph, every minimal crossing edge belongs to some MST of the graph and every MST contains a minimal crossing edge
    - (cycle property) for a graph, G, an MST of the graph G', obtained by adding an edge e to G, can be found by adding e to an MST of G and deleting a maximal edge on the resulting cycle
    - Prim's algorithm computes an MST of any connected graph
    - Kruskal's algorithm computes an MST of any connected graph
    - Boruvka's algorithm computes the MST of any connected graph
- if the graph's edge weights are distinct it has a unique MST
- no requirement that the minimal edge be the only MST edge connecting the two sets
- the cut property is the basis for finding MSTs, and it can also serve as an optimality condition that characterizes MSTs
- Prim's algorithm:
    - start with any vertex as a single-vertex MST
    - add V - 1 edges to it
        - choose a minimal edge that connects a vertex on the MST to a vertex not yet on the MST

- Kruskal's algorithm:
  - process edges in order of their length, shortest first
  - add to the MST each edge that does not form a cycle with a previously added edge
  - stop after V - 1 edges have been added
- Boruvka's algorithm:
  - add each edge that connects eachvertex to its closest neighbor
  - if edge weights are distinct, creates a forest of MST subtrees
  - add an edge to each tree that connects one tree with its closest neighbor (a minimal edge connecting any vertex in one tree with any vertex in another)
  - iterate until the forest is one tree
- generalized algorithm:
  - begin with a forest of single-vertex MST subtrees (with no edges)
  - add a minimal edge connecting any two subtrees in the forest
  - continue until V - 1 edges have been added and a single MST remains
- constituent operations:
  - find a minimal edge connecting two subtrees
  - determine whether adding an edge would create a cycle
  - delete the longest edge on a cycle

Prim's Algorithm and Priority-First search

- simplest and method of choice for dense graphs
- general process:
  - maintain a cut of vertices in the tree and those not yet in the tree
  - start by putting any vertex on the MST
  - then put a minimal crossing edge on the MST
  - repeat V - 1 times
- generally just need to keep track of the minimum edge and check whether vertex addition requires update of the minimum using data structures that store the following:
  - tree edges
  - shortest edge connecting each nontree vertex to the tree
  - the length of that edge
- after adding a new edge:
  - check to see whether adding the new edge brought any nontree vertex closer to the tree
  - find the next edge to add to the tree
- Prim's algorithm is a general graph search, using a fringe that is a minimum priority queue:
  - move a minimal edge from the fringe to the tree
  - visit the vertex that it leads to
  - put onto the fring any edges that lead from that vertex to a nontree vertex
  - replace the longer edge when two edges on the fringe point to the same vertex
- priority-first search (PFS):
  - generalized graph search with a priority queue as the fringe

- Prim's algorithm == PFS with edge weights for priorities
- properties:
  - using Prim's algorithm we acan find the MST of a dense graph in linear time
  - using a PFS implementation of Prim's algorithm that uses a heap for the priority queue, an MST can be computed in E * lg(V) time
  - for all graph and all priority functions, we can compute a spanning tree with PFS in linear time plus time proportional to the time required for V insert, V delete the minimum and E decrease key operations in a priority queue of size at most V

```
// Program 20.6 Prim's MST algorithm
```

```
// Priority-first search
```

- many other algorithms differ only in choice of priority queue
- discovered by Jarnik, and later Prim and Dijkstra independently (sometimes referred to by each name)

Kruskal's Algorithm

- both Prim's and Kruskal's build the MST one edge at a time, but Kruskal's finds an edge that connects a tree in a spanning forest
- can terminate in one of two ways, once V - 1 edges have been found, either an MST has been built or the graph is not connected
- properties:
  - Kruskal's algorithm computes the MST of a graph in time proportional to E * lg(E)
  - a priority queue based version of Kruskal's algorithm computes the MST of a graph in time proportional to E + X*lg(V) where X is the number of graph edges not longer than the longest edge in the MST

```
Program 20.8 Kruskal's MST algorithm
```

- typically the cost of finding an MST with Kruskal's algorithm is lower than the cost of processing all edges because the MST is complete before this
- can also apply quicksort to Kruskal's algorithm to speed up the algorithm

Boruvka's Algorithm

- oldest MST algorithm
- can be implemented using the union-find ADT
- process:
  - maintain a vertex indexed vector that identifies for each MST subtree the nearest neighbor

    - for each edge:
        - discard it if it connects two vertices in the same tree
        - otherwise check the nearest neighbor distances between the two trees that the edge connects and update thme if appropriate

```
// Prgoram 20.9 Boruvka's MST algorithm
```

- three factors that make the implementation efficient:
    - cost of each find is essentially constant
    - each stage decreases the number of MST subtrees in the forest by at least a factor of 2
    - a substantial number of edges is discarded during each stage
- properties:
    - the running time of Boruvka's algorithm for computing the MST of a gaph is O(E * lg(V) * lg(*E))
- can remove the lg(*E) factor to a running time of E * lg(V) by representing MST subtrees with doubly-linked lists but makes the implementation more complicated

Comparisons and Improvements

```
Cost of MST algorithms
Prim(standard)         V^2             optimal for dense graphs
Prim(PFS,heap)         E*lg(V)         conservative upper bound
Prim(PFS,d-heap)       E*log sub d(V)  linear unless extremely sparse
Kruskal                E*lg(E)         sort cost dominates
Kruskal(partial sort)  E+X*lg(V)       cost depends on longest edge
Boruvka                E*lg(V)         conservative upper bound
```

- properties:

    - given a graph with V vertices and E edges, let d denote the density E/V. If d < 2 then the running time of Prim's algorithm is proportional to Vlg(V). Otherwise, we can improve the worst case running time by a factor of lg(E/V) by using an E/V-ary heap for the priority queue

- priority queue options:

    - Fibonacci heap (extension of binomial queue)
    - binomial queues
    - radix methods
    - multiway heap

```
// Program 20.10 multiway heap priority queue implementation
```

Euclidean MST

- Euclidean MST: given a set of points in a plane, find the shortest set of lines connecting all of the points
- can use Prim's algorithm in N^2 time (N vertices and N * (N-1) / 2 edges) but this is generally too slow
- properties:
  - the Euclidean MST of N points can be found in time proportional to N * log(N)
  - finding the Euclidean MST of N points is no easier than sorting N numbers
- Delauney triangulation: a planar graph whose number of edges is proportional to N that contains the MST

Shortest Paths

- networks: weighted digraphs
- use pointers to abstract edges for weighted digraphs
- when working with networks, it is convenient to add self-loops to each representation
- all connectivity properties of digraphs are relevant for networks
- definitions:
  - shortest path: a directed simple path from two vertices, s and t, in a network such that no other such path has a lower weight
- problems:
  - source-sink shortest path: the shortest path between a start (source) and end (sink) vertex
  - single-source shortest paths: the set of shortest paths from start vertex to all other vertices in a graph
  - all-pairs shortest paths: the set of shortest paths connecting each pair of vertices in the graph
- real world relationships:
  - road maps
  - airline routes

Underlying Principles

- relaxation: the process of moving through a graph, gathering information, and at each step, testing whether there exists some path that is shorter than a known path

- edge relaxation: an operation that tests whether traveling along a given edge gives a new shortest path to its destination vertex

- path relaxation: an operation that tests whether traveling through a given vertex gives a new shortest path connecting two other given vertices

- components:
    - a vertex indexed vector that contains the shortest known paths from the start vertex to to each vertex
    - a vertex indexed vector that records the last edge on a shortest path from the source to the indexed vertex (constitutes a tree)
- definitions:
    - shortest-paths tree: given a network and a designated vertex, s, the subnetwork containing s and all of the vertices reachable from s that form a directed tree rooted at s such that every tree path is a shortest path in the network
- note that there may be multiple paths of the same length, i.e. SPTs are not necessarily unique
- need to initialize the vertex indexed vector of weights with values that will not cause overflow
- properties:
    - if a vertex x is on a shortest path from s to t, then that path consists of a shortest path from s to x followed by a shortest path from x to t
- each algorithm has a direct corollary to algebraic concepts
- need to implement an abstract + operator and < operator
- each use the following operations:

```
// single-source implementations
Edge *path_r(int w) const { return spt[w]; }
double dist(int v) { return wt[v]; }
// all paths implementations
Edge *path(int w) { return p[s][t]; }
double dist(int v) { return d[s][t]; }
```

Dijkstra's Algorithm

- general process (similar to Prim's algorithm):
    - put source on SPT
    - build SPT one edge at a time, choosing the edge that gives the shortest path from the source to an edge not yet on the tree
- properties:
    - Dijkstra's algorithm solves the single-source shortest paths problem in that have non-negative weights
    - any SPT in a dense network can be found with Dijkstra's algorithm in linear time

    - PFS can compute a spanning tree for all networks and all priority functions in time proportional to the time required for V insert, V delete the minimum, and E decrease key operations in a priority queue of size at most V
    - Dijkstra's algorithm that uses a heap for the priority queue implementation can compute any SPT in time proportional to E * lg(V)
    - given a graph with density d (E/V), if d < 2, then the running time of Dijkstra's algorithm is proportional to V*lg(V). Otherwise, the worst case running time can be improved by a factor of lg(E/V) to O(E * lg sub d(V)) (which is linear if E is at least V^(1+E)) by using an E/V-ary heap for the priority queue
- Dijkstra's algorithm also solves the source sink shortest paths problem if we stop when the sink comes off the priority queue
- Dijkstra's algorithm is a PFS graph search scheme

```
// Dijkstra's algorithm (priority-first search)
```

```
algorithm       priority            result
DFS             reverse preorder    recursion tree
BFS             preorder            SPT(edges)
Prim            edge weight         MST
Dijkstra        path weight         SPT
```

- the PFS implementation permutations are described here (before table 21.2)

All-Pairs Shortest Paths

- can implement all-pairs shortest paths with two main methods:
    - run Dijkstra's algorithm for each vertex
    - run a modification of Warshall's algorithm called Floyd's algorithm

```
// Program 21.2 All-pairs shortest-paths ADT
```

- weighted diameter: for a network, the longest of the shortest path lengths for each vertex

```
// Program 21.3 Computing the diameter of a network
```

- each algorithm uses space proportional to V^2 for private data members to support constant time queries

- properties:

    - with Dijkstra's algorithm, we can find all shortest paths in a network that has non-negative weights in time proportional to V*E log sub d(V) where d = 2 if E < 2V and d = E/V otherwise

  - with Floyd's algorithm we can find all shortest paths in a network in time proportional to V^3

```
// Program 21.4 Dijkstra's algorithm for all shortest paths
```

```
// Program 21.5 Flod's algorithm for all shortest paths
```

- Dijkstra's method is the method of choice for sparse networks with a running time close to V * E
- Floyd's algorithm is more competitive as density increases, with time proportional to V^3
- Floyd's algorithm is widely used because it is easy to implement

Shortest Paths in Acyclic Networks

- the shortest paths problems can be solved with more ease and simplicity for DAGs
- may refer to weighted DAGs as acyclic networks
- four basic ideas from ch 19 are more effective for DAGs:
  - use DFS to solve the single-source problem
  - use a source queue to solve the single-source problem
  - invoke each method once for each vertex to solve the all-pairs problem
  - use a single DFS with dynamic programming to solve the all-pairs problem
- multisource shortest paths: given a set of start vertices, the set of shortest paths for each start vertex to a separate specified vertex, w
- properties:
  - we can solve the multisource shortest-paths problem and the multisource longest paths problem in acyclic networks in linear time
  - we can solve the all-pairs shortest-paths problem in acyclic networks with a single DFS in time proportional to V*E

```
// Program 21.6 Longest paths in an acyclic network
```

```
// Program 21.7 All shortest paths in an acyclic network
```

Euclidean Networks

- Euclidean networks: networks whose vertices are points in a plane and whose edge weights are defined by the geometric distances between the points

- there is a fast algorithm for the source-sink shortest-path problem in Euclidean networks
- Euclidean networks properties:
  - distances satisfy the triangle inequality
  - triangle inequality: distance from s to d is never greater than the distance from s to x plus distance from x to d
  - often symmetric -> edges run in both directions
- implement using a standard PFS version of Dijkstra's
  - initialize `wt[s]` to dist(s, d) instead of 0.0
  - use a priority of `wt[v] + e_.wt() + distance(w, d) - distance(v, d)`
  - the Euclidean heuristic maintains the invariant that `wt[v] - distance(v, d)` is the length of the shortest path through the network s to v for every tree vertex, i.e. `wt[v]` is a lower is a lower bound on the length of the shortest possible path through v from s to d
- properties:
  - PFS with the Euclidean heuristic solves the source-sink shortest paths problem in Euclidean graphs
- the precise savings of the Euclidean heuristic is dependent on the nature if the input graph and is not guaranteed
- the property above applies for any function that gives a lower bound on the distance from each vertex to d
- the Euclidean heuristic is a specific instance of an A* algorithm

Reduction

- the shortest paths problem represents a mathematical model that can be used to solve many other problems that are not graph related

- definitions:

  - reduction: one problem reduces to another if an algorithm that solves one can be used to solve another in a no more than a constant multiple of the worst case running time for the original
  - equivalence: two problems are equivalent if they reduce to each other
  - infeasible: a problem instance that does not have a solution

- properties:

  - the transitive closure problem reduces to the all-pairs shortest paths problem with non-negative weights
  - in networks with non constraints on edge weights, the longest-path and shortest-path problems (single-source or all-pairs) are equivalent
  - the job-scheduling problem reduces to the difference constraints problem
  - the difference constraints problem with positive constants is equivalent to the single-source longest-paths problem in an acyclic network

  - the job scheduling with deadlines problem reduces to the shortest-paths problem (with negative weights allowed)
  - a problem is NP-hard if there is an efficient reduction to it from any NP-hard problem
  - in networks with edge weights that could be negative, shortest paths problems are NP-hard

- job scheduling:

  - set of precedence relations specify an ordering of jobs and multiple jobs can be worked on at the same time
  - determine the minimum amount of time required to complete all the jobs while satisfying all of the precedence constraints

```
// Program 21.8 Job scheduling
```

- difference constraints:

  - assign non-negative values on a set of variables x sub 0 to x sub n that minimize the value of x sub n while satisfying a set of difference constraints on the variables, each of which specifies that the difference between two of the variables must be greater than or equal to a given constant

- linear programming:

  - assign non-negative values on a set of variables x sub 0 to x sub n that minimize the value of the value of the specified linear combination of the variables, subject to a set of constraints on the variables, each of which specifies that a given linear combination of the variables must be greater than or equal to a given constant

- job scheduling with deadlines:

  - adds the constraint to the job scheduling problem that jobs must start within a given amount of time of the start job

- NP-hard:

  - no efficient algorithm is known that is guaranteed to solve the problem
  - there is not a good possibility of solving the problem

- see table 21.3 for a table of reduction implications with upper bounds and lower bounds relationships

- PERT chart: a network representation for job-scheduling problems with jobs as edges

Negative weights

- TODO: add more info here
- negative weights complicate shortest paths problems

- low-weight shortest paths tend to have more edges than higher weight paths
- shortest paths in networks with no negative cycles:
    - given a network that may have negative weights but no negative cycles solve the shortest path problem, single-source shortest path problem, and all-pairs problem
- negative cycle detection
- arbitrage
- Bellman-Ford algorithm:
    - solves the single-source shortest paths problem in networks that contain no negative cycles
    - TODO: explain here
- reweighting:
    - to reweight an edge, add to that edge's weight the difference between the weights of the edge's source and destination
    - to reweight a network, reweight all of that network's edges
- Johnson's algorithm:
    - TODO: explain here
- properties:
    - the job scheduling with deadlines problem reduces to the shortest-paths problem in networks that contain no negative cycles
    - Floyd's algorithm solves the negative cycle detection problem and the all-pairs shortest paths problem in networks that contain no negative cycles in tim eproportional to V^3
    - the Bellman-Ford algorithm solves the single-source shortest paths problem in networks that contain no negative cycles in time proportional to V*E
    - the Bellmain-Ford algorithm can solve the negative cycle detection problem in time proportional to V*E
    - reweighting a network does not affect its shortest paths
    - in any network with no negative cycles, pick any vertex s and assign to each vertex v a weight equal to the length of a shortest path to v from s. Reweighting the network with these vertex weights yields non-negative edge weights for each edge that connects vertices reachable from s
    - Johnson' algorithm solves the all-pairs shortest paths problem in networks that contain no negative cycles in time proportional to V*E log sub d(V), where d = 2 if E < 2V, and d = E/V otherwise

```
// Program 21.9 Bellman-Ford algorithm
```

- summary for all-pairs shortest paths in networks with negative weights but no negative cycles:
    - use Bellmain-Ford to find a shortest paths forest in the original network
    - if the algorithm detects a negative cycle, report and terminate
    - reweight the network from the forest
    - apply the all-pairs version of Dijkstra's algorithm to the reweighted network

Perspective

```
Table 21.4 of shortest-paths algorithms
                weight constraint           algorithm           cost            co
single-source
                non-negative                Dijkstra            V^2             op
                non-negative                Dijkstra (PFS)      E*lg(V)         co
                acyclic                     source queue        E               op
                no neg cycles               Bellman-Ford        V*E             ca
                none                        open                ?               NP

all-pairs
                non-negative                Floyd               V^3             sa
                non-negative                Dijkstra (PFS)      V*E*lg(V)       co
                acyclic                     DFS                 V*E             sa
                no neg cycles               Floyd               V^3             sa
                no neg cycles               Johnson             V*E*lg(V)       co
                none                        open                ?               NP
```

- shares general properties with transitive closure in digraphs
- entry point to network-flow problems and linear programming

Network Flow

- networks with dynamic material flowing through edges with different costs for different routes
- have many equivalent problems, i.e. reductions from one to another and many good flow solutions have been created
- distribution problems (movement from one vertex to another):
    - merchandise distribution: distribution from factories to distribution outlets with varying capacities and varying costs. Find a way to make supply meet demand at least cost
    - communications: requests from servers between channels at varying rates of exchange. Find the maximum rate between two servers. Find the cheapest way to send information.
    - traffic flow: many related questions
- matching problems (connecting one vertex to another):
    - job placement
    - minimum distance point matching: with two sets of N points, find the set of N line segments, with one endpoint from each point, with the minimum total length
- cut problems (remove edges to cut network into two or more pieces):
    - network reliability
    - cutting supply lines
- consider minflow and maxflow problems
- many real world problems can be reduced to these graph and network problems but it takes time to realize how and effort to determine if they are intractable or solvable

Flow Networks

- aids intuition to imagine a set of fluid pipes with valves at each end with a single source and single sink, each vertex has a flow equilibrium, and consider flow and capacity
- can model with a network (weighted digraph) with a single source and single sink
- related to fluid flow, distribution, communications, traffic flow, etc.
- definitions:
    - st-network: a network with a designated source, s, and a designated sink, t
    - flow network: an st-network with positive edge weights
    - capacities: the weights of edges in a flow network
    - flow: a set of non-negative edge weights with no flow larger than the edge's capacity and total flow into a vertex == total flow out
    - edge flows: non-negative edge weights
    - circulation: a flow between s and t that is set to equal the total network flow
    - uncapacitated: edges with infinite capacity
- maximum flow (maxflow):
    - given an st-network, find a flow such that no other flow from s to t has a larger value
- properties:
    - any st-flow has the property the property that outflow from s is equal to the inflow to t
    - the value of the flow for the union of two sets of vertices is equal to the sum of the values of the flows for the two sets minus the sum of the weights of the edges that connect a vertex in one to a vertex in the other
    - (Flow decomposition theorem) Any circulation can be represented as flow along a set of at most E directed cycles
    - any st-network has a maxflow such that the subgraph induced by non-zero flow values is acyclic
    - any st-network has a maxflow that can be represented as flow along a set of at most E directed paths from s to t
- use a standard Graph (typically sparse, sparse multigraph) representation with pointers to an Edge class that contain pcap and pflow (private capacity and private flow) members with public getters
- assume all weights are m bit integers to test for equality among linear combinations and running times can depend on the relative value of weights
- use M = 2^m for m-bit integers where the ratio of the largest to smallest non-zero flow is M

```
// Program 22.1 Flow check and value computation
```

Augmenting Path Maxflow Algorithms

- TODO: more work in the running time analysis at the end of this section

- Ford-Fulkerson (augmenting-path method) - method for increasing flows incrementally along paths from source to sink
- simple process:
    - given a minimum of unused capacities of edges on a path, x, increase the flow of each edge along that path
    - continue until all paths from source to sink have at least one full edge (an edge at max capacity)
- improvement:
    - classify forward edges and backward edges (from source to sink, or vice versa)
    - for any path with no full forward edges and no empty backward edges, increase flow by increasing forward edges and decreasing backward edges
- Ford-Fulkerson:
    - start with zero flow everywhere
    - increase the flow along any path from source to sink with no full forward edges or empty backward edges
    - continue until there are no such paths in the network
- Ford-Fulkerson leads to maximal flow value and can be shown by the maxflow-mincut theorem
- definitions:
    - st-cut: a cut that places vertex s in one of its sets and vertex t in the other
    - cross edge: an edge that connects a vertex in one set with a vertex in the other
    - cut set: set of crossing edges
    - st-cut capacity: sum of the st-cut's st-edges
    - flow across: difference between the sum of the st-edges and the sum of the ts-edges
    - residual network: for a flow network and a flow, the network produced if the flow has the same vertices as the original and one or two edges in the residual network for each edge in the original, defined as follows:
        - for each edge, v-w, in the original, let f be the flow capacity
        - if f is positive, include an edge w-v in the residual with capacity f
        - if f is less than c, include an edge v-w in the residual with capacity c-f
- properties:
    - for any st-flow, the flow across each st-cut is equal to the value of the flow
    - no st-flow's calue can exceed the capacity of any st-cut
    - IMPORTANT!!!
    - maxflow-mincut theorem: the maximum value among all st-flows in a network is equal to the minimum capacity among all st-cuts

  - for a maximum edge capacity, M, the number of augmenting paths needed by any implementation of the Ford-Fulkerson algorithm is at most equal to VM
  - the time required to find a maxflow if O(VEM), which is O(V^2M) for sparse networks
  - the number of augmenting paths needed in the shortest-augmenting path implementation of the Ford-Fulkerson algorithm is at most V*E/2
  - the time required to find a maxflow in a sparse network is O(V^3)
  - the number of augmenting paths needed in the maximal-augmenting path implementation of Ford-Fulkerson is at most `2*E*lg(M)`
  - the time required to find a maxflow in a sparse network is O(V^2*lg(M)*lg(V))
- minimum cut: given an st-network, find an st-cut such that the capacity of no other cut is smaller

```
// Program 22.2 Flow-network edges
```

```
// Program 22.3 Augmenting-paths maxflow implementation
```

```
// Program 22.4 PFS for augmenting-paths implementation
```

- Ford-Fulkerson has many implementations based on various research efforts
- simplest (Edmonds & Karp's shortest augmenting path) uses a queue for the fringe
- another (Edmonds & Karp's maximum-capacity augmenting path)
  - augment along the path that increases the flow by the largest amount using this for the priority
- difficulties in measuring performance -> many versions depend on V, E, and the edge capacities

Preflow-Push Maxflow Algorithms

- incrementally move flow along outgoing edges of vertices that have more inflow than outflow
- Goldberg and Tarjan
- definition:
  - preflow: a set of positive edge flows satisfying the conditions that the flow on each edge is no greater than tha edge's capacity and that inflow if no smaller than outflow for every internal vertex
  - active vertex: internal vertex whose inflow is larger than its outflow (by convention, the source and sink are never active)
  - height function: a set of non-negative vertices h(0)...h(V-1) for a given flow in a flow network such thath(t) = 0 for the sink t and h(u) <= h(v)+1 for every edge u-v in the residual network for the flow
  - eligible edge: an edge u-v in the residual network with h(u) = h(v)+1

- properties:
    - for any flow and associated height function, a vertexs height is no larger than the length of the shortest path from that vertex to the sink in the residual network
    - if a vertex's height is greater than V, then there is not path from that vertex to the sink in the residual network
    - the edge-based preflow-push algorithm preserves the validity of the height function
    - while the preflow-push algorithm is in execution on a flow network, there exists a (directed) path in that flow network's residual network from each active vertex to the source, and there are no (directed) paths from source to sink in the residual network
    - during the preflow-push algorithm, vertex heights are always less than 2*V
    - the preflow-push algorithm computes a maxflow
    - the worst-case running time of the FIFO queue implementation of the preflow-push algorithm is proportional to V^2*E

```
// Program 22.5 Preflow-push maxflow implementation
```

Maxflow Reductions

- TODO: more description of the problems here

- consider a number of maxflow reductions to show its general applicability

- problems:

    - maxflow in general networks: the flow in a network that maximizes the total outflow from its sources (and the total inflow to its sinks)
    - vertex-capacity constraints: find maxflow satisfying additional constraints specifying that the flow through each vertex must not exceed som fixed capacity
    - acyclic networks: maxflow in an acyclic network
    - maxflow in undirected networks:
        - undirected flow networks: a weighted graph with integer edge weights that are interpreted as capacities
        - circulation (in undirected flow networks): an assignment of weights and directions to the edges satisfying the conditions that the flow on each edge is no greater than that edge's capacity and that the total flow into each vertex is equal to the total flow out of that vertex
        - find a circulation that maximizes the flow in a specified direction in a specified edge
    - feasible flow:
    - maximum-cardinality bipartite matching:
    - edge connectivity:
    - vertex connectivity:

- properties:
  - the maxflow problem for general networks is equivalent to the maxflow problem for st-networks
  - the maxflow problem for flow networks with capacity constraints on vertices equivalent to the standard maxflow problem
  - the maxflow problem for acyclic networks is equivalent to the standard maxflow problem
  - the maxflow problem for undirected st-networks reduces to the maxflow problem for st-networks
  - the feasible flow problem reduces to the maxflow problem
  - the bipartite-matching problem reduces to the maxflow problem
  - the time required to find a maximum-cardinality matching in a bipartite graph is O(V*E)
  - (Menger's Theorem) the minimum number of edges whose removal disconnects two vertices in a digraph is equal to the maximum number of edge-disjoint paths between the two vertices
  - the time required to determine the edge connectivity of an undirected graph is O(E^2)

```
// Program 22.6 Feasible flow via reduction to maxflow
```

```
// Program 22.7 Bipartite matching via reduction to maxflow
```

Mincost Flows

- many solutions for a given maxflow problem
- one with the fewest edges is mincost flow which are generally more difficult
- definitions:
  - cost: the sum of the flow costs of that flows edges
  - flow cost: the product of an edge's flow and cost for an edge in a flow network
  - residual network: for a flow network and a flow, the network produced if the flow has the same vertices as the original and one or two edges in the residual network for each edge in the original, defined as follows:
    - for each edge, v-w, in the original, let f be the flow capacity
    - if f is positive, include an edge w-v in the residual with capacity f
    - if f is less than c, include an edge v-w in the residual with capacity c-f
- mincost maxflow: the maxflow for a flow network with a minimum cost

- mincost feasible flow: a flow is feasible if the vertex weights (supply if positive, demand if negative) if the sum of the vertex weights is negative and the difference between each vertex's outflow and inflow. The problem finds a feasible flow of minimum cost.

```
// Program 22.8 Computing flow cost
```

- properties:
    - the mincost-feasible flow problem and the mincost maxflow problems are equivalent
    - a maxflow is a mincost maxflow if and only if its residual network contains no negative cost (directed) cycle
    - the number of augmenting cycles needed in the generic cycle canceling algorithm is less than ECM
    - the time required to solve the mincost-flow problem in a sparse network is O(V^3CM)
- cycle canceling algorithm:
    - find a maxflow
    - augment the flow along any negative-cost cycle in the residual network
    - continue until none remain

```
// Program 22.9 Cycle canceling
```

Network Simplex Algorithm

- TODO: more detail of the implementation and development

- the running time of the cycle canceling algorithm is based on the number of negative cost cycles, the flow cost, and the time to find the cycles

- the network simplex algorithm drastically reduces cost for finding the cycle

- definitions:

    - feasible spanning tree: any spanning tree of the network that contains all of the flow's partial edges for a maxflow with no cycle of partial edges
    - reduced cost: given a flow in a flow network with edge costs, let c(u, v) denote the cost of u-v in the flow's residual network. For any potential function, P, the reduced cost of an edge u-v in the residual network with respect to P, denoted by c(u, v), is defined as c(u, v) - (P(u) - P(V))
    - valid (set of vertices): if all tree edges have zero reduced cost
    - eligible (nontree edge): a nontree edges is eligible if the cycle that it creates with tree edges is a negative-cost cycle in the residual network

- network simplex algorithm:

    - with respect to any flow, each edge u-v is in one of three states:
        - empty: so flow can be pushed from only u to v

        - full: so flow can be pushed from only v to u
        - partial (neither empty or full): so flow can be pushed either way
    - build a spanning tree
        - option 1 - build a maxflow, break cycles of partial edges by augmenting along each cycle to fill or empty one of its edges, then add full or empty edges to the remaining partial edges to make a spanning tree
        - option 2 - start with a maxflow in a dummy edge from source to sink, then, using the edge as the only possible partial edge, build a spanning tree with any graph search
    - at this point, adding any edge creates a cycle
    - maintain a vertex-indexed vector (spanning tree) and compute the reduced cost an edge and set the value of the vertex potentials such that all tree edges have zero reduced cost

- properties:
    - an edge is eligible if and only if it is a full edge with positive reduced cost or an empty edge with negative reduced cost
    - if we have a flow and a feasible spanning tree with no eligible edges, the flow is a mincost flow
    - if the generic network simplex algorithm terminates, it computes a mincost flow

- choosing the spanning tree representation:
    - related computations:
        - computing the vertex potentials
        - augmenting along the cycle
        - identifying an empty or a full edge on it
        - inserting a new edge and removing an edge on the cycle formed

```
// Program 22.10 Vertex potential calculation
```

```
// Program 22.11 Augmenting along a cycle
```

```
// Program 22.12 Spanning tree substitution
```

```
// Program 22.13 Eligible edge search
```

```
// Program 22.14 Network simplex (basic implementation)
```

```
// Program 22.15 Network simplex (improved implementation)
```

Mincost-Flow Reductions

- TODO: more detail here
- problems:
    - transportation:
    - assignment:
    - mail carrier:
- properties:
    - the single-source shortest paths problem (in networks with no negative cycles) reduces to the mincost-feasible flow problem
    - in mincost-flow problems, we can assume, without loss of generality, that edge costs are non-negative
    - the transportation problem is equivalent to the mincost-flow problem
    - the assignment problem reduces to the mincost-flow problem
    - the mail carrier's problem reduces to the mincost-flow problem

Perspective

- end study with network flow because:
    - the network flow model validates the practical utility of the graph abstraction in countless applications
    - the maxflow and mincost-flow algorithms are natural extensions of graph algorithms for simpler problems
    - the implementations exemplify the important role of fundamental algorithms and data structures in achieving good performance
    - the maxflow and mincost-flow models illustrate the utility of the approach of developing increasingly general problem solving models and using them to solve broad classes of problems
- problems:
    - maximum matching: the subset of edges in which no vertex appears more than once and whose total weight is such that no other such set of edges has a higher total weight (for a graph with edge weights)
    - multicommodity flow: computing a second flow such that the sum of an edge's two flows is limited by that edge's capacity, both flows are in equilibrium, and the total cost is minimized
    - convex and nonlinear costs: many network flow applications call for non-linear or more complex cost functions and combinations of variables
    - scheduling: much more complex and generally useful scheduling problems
- general problem solving algorithms:
    - shortest paths

    - maxflow
    - mincost-flow
- generic algorithms:
    - Bellman-Ford
    - more

NOTES:

- st in the search implementations means search tree and is the parent-link representation of the search tree
- st in st-network stands for source-sink where s = source and t = sink
- edge classification:
    - tree edge:
        - an edge corresponding to a recursive call
        - edge that is visited for the first time the first time its destination vertex is encountered
        - link from v-w is a tree link if w is unmarked
        - link from v-w is a parent link if `st[w]` is v
        - each tree edge in a graph corresponds to a tree link and a parent link in a DFS tree
        - tree edges are tree links on the first encounter and parent links on the second
    - back edge:
        - edge connecting a vertex with an ancestor in its DFS tree that is not its parent
        - an edge that is visited for the first time and its destination vertex has been encountered and it is an ancestor of the source
        - a link from v-w is a back link if `ord[w] < ord[v]`
        - a link from v-w is a down link if `ord[v] < ord[w]`
        - each back edge in a graph corresponds to a back link and a down link in a DFS tree
        - back edges are back links on the first encounter and down links on the second encounter
        - in a DFS forest for a digraph, a back edge is an edge to a visited node with a higher postorder number
    - forward edge: an edge that is visited for the first time and its destination vertex has been encountered and it is a descendant of the source
    - cross edge:
        - an edge visited for the first time that is neither a back edge or a forward edge

        - in a DFS forest for a digraph, a cross edge is an edge to a visited node with a lower preorder number
    - down edge:
        - in a DFS forest for a digraph, a down edge is an edge to a visited vertex with a higher preorder number
    - NOTE: parent links and back links both have order[w] < order[v], so if search_tree[w] is not v means v-w is a back link
- by contrast with undirected graphs, DFS on a digraph does not give full information about reachability from any vertex other than the start vertex because tree edges are directed and the search structures have cross edges (cannot assume there is a way to get back)
- simple pseudocode:

```
DFS(u)
for each neighbor v of u
  if v is unvisited, tree edge, DFS(v)
  else if v is explored, bidirectional/back edge
  else if v is visited, forward/cross edge
```

- from "Python Algorithms"

```
In describing traversal, I have distinguished between three kinds of nodes:
those we don’t know about,
those in our queue,
and those we’ve visited (and whose neighbors are now in the queue).

Some books (such as Introduction to Algorithms, by Cormen et al., mentioned in Cha
introduce a form of node coloring, which is especially important in DFS.

Each node is considered white to begin with;
they’re gray in the interval between their discover time and their finish time,
and they’re black thereafter.

You don’t really need this classification in order to implement DFS,
but it can be useful in understanding it...

In terms of Trémaux’s algorithm,
gray intersections would be ones we’ve seen but have since avoided;
black intersections would be the ones we’ve been forced to enter a second time
(while backtracking).

These colors can also be used to classify the edges in the DFS tree.

If an edge uv is explored and the node v is white,
the edge is a tree edge—that is, it’s part of the traversal tree.

If v is gray, it’s a so-called back edge, one that goes back to an ancestor
in the DFS tree.
```

```
Finally, if v is black, the edge is either what is called a forward edge or a
cross edge.

A forward edge is an edge to a descendant in the traversal tree,
while a cross edge is any other edge (that is, not a tree, back or forward edge).

Note that you can classify the edges without actually using any explicit color
labeling.

Let the time span of a node be the interval from its discover time to its finish t

A descendant will then have a time span contained in its ancestor’s,
while nodes unrelated by ancestry will have nonoverlapping intervals.

Thus, you can use the timestamps to figure out whether something is, say,
a back or forward edge.

If your timestamps are incremented in single steps,
a child interval (forward edge) will be “just inside” the parent’s interval.
Even with a color labeling, you’d need to consult the timestamps to differentiate
between forward and cross edges.

You probably won’t need this classification much, although it does have one import

If you find a back edge, the graph contains a cycle, but if you don’t, it doesn’t.
In other words, you can use DFS to check whether a graph is a DAG
(or, for undirected graphs, a tree).
```

- need to be careful in use of graph.adjacent() and pushing to queue:
    - if the call returns edges and the queue contains edges, make sure to visit the destination edge
    - if the call returns edges and the queue contains vertex indices, push the destination vertex
    - if the call returns vertex indices and the queue contains edges, make sure to push a constructed edge with proper source/dest