Data Structures and Algorithms — CSci 230
Chapter 9 — Graphs and Graph Algorithms

Overview

- We will cover up through section 9.6, skipping Section 9.4.
- Graphs are used to model and solve problems in a variety of domains, including networking and routing, scheduling, optimization.

Definitions

- A graph $G = (V, E)$ consists of a set of vertices, V, and a set of edges, E. Each edge in $e \in E$ is a pair, $e = (u, v)$, where $u \in V$ and $v \in V$ (or, more succinctly, $u, v \in V$).
  - In practice, the vertices and edges will have attributes and meanings associated with them. These would be stored in a class object defined for vertices and a class object defined for edges.
- We will define a variety of terms in class:
  - Directed and undirected graphs.
  - Subgraphs.
  - Edge weights.
  - Paths, simple paths, and path lengths.
  - Cycles.
  - Acyclic graphs and directed acyclic graphs.
  - Connected graphs.
  - Strongly and weakly connected (directed) graphs.
  - Complete graphs.
  - Trees.
- Notions to remember and conventions we will use in drawing and discussing graphs:
  - Vertices will be "labeled" from 1 to $|V|$ (or from 0 to $|V| - 1$), and drawn as circles with numbers inside them.
  - Weighted edges will be drawn with numbers on them indicating their weights. Directed edges will be drawn with arrows, undirected edges without arrows.
  - Except in special situations, there is no spatial information associated with a graph, and the numbering of vertices is arbitrary. Therefore, the same graph can be drawn and labeled in many different ways. (In some applications there will be x, y coordinates associated with graph vertices.)

Graph representations

- Adjacency lists.
- Adjacency matrices.

Exercise

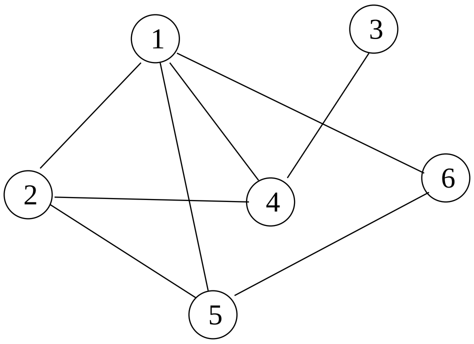
1. For each graph shown,

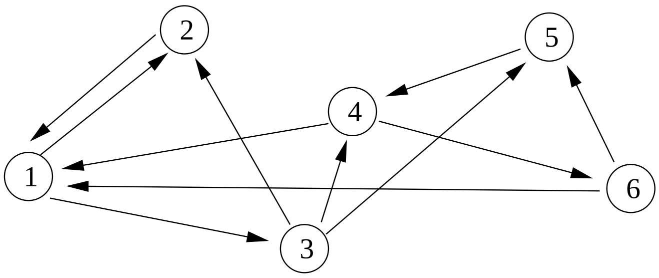
   (a) give the adjacency list representation,

   (b) give the adjacency matrix representation,

   (c) tell if the graph is connected, and

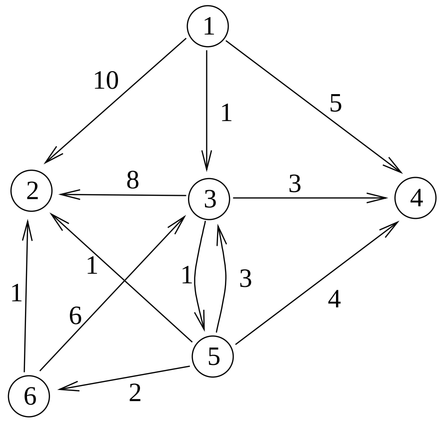
   (d) find the longest cycle.





Shortest Path Problems — Dijkstra's Algorithm

- **Problem:** Given a directed graph $G = (V, E)$, such that there is a positive weight associated with each edge, and given a vertex $v_0 \in V$, for each $v \in V$, find the path with the least weight (the "shortest path") from v_0 to v.
- This problem has many applications.
- Here is a graph we will use to study the problem:



- Dijkstra's algorithm is an example of a *Greedy* algorithm.

Pseudo-code

To help make the algorithm more understandable, here is a substantial revision of the author's code. Assume that in the graph, V is an array of vertices (type `Vertex`). Each vertex will store a list of its edges, and each edge will represented by a vertex index and a weight in the following structure

```
struct AdjVertex
{
    int W;          //  index of the neighboring vertex
    float Weight;   //  edge weight
};
```

Next, as discussed in outlining the algorithm, a certain amount of additional information for each vertex, specific to Dijkstra's algorithm rather than to a general graph, must be stored in an auxiliary `PathTable`. Here is a structure to store this information

```
struct PathTableEntry
{
    bool  PathKnown;   //  has the shortest path been found?
    float Distance;    //  length of current shortest path
    int   PathPred;    //  predecessor on the shortest path
};
```

Now, here is a function to initialize the `PathTable` for a given graph.

```
void
InitTable( int StartVertex, const Graph& G,
           PathTableEntry T[] )  // assumes the table is pre-allocated
{
   for ( int i=0; i<G.NumVertices(); i++ ) {
      T[i].PathKnown = false;
      T[i].Distance  = Infinity;   //  assume this is defined
      T[i].PathPred  = -1;
   }
   T[StartVertex].Distance = 0;
}
```

Finally, here is the algorithm

```
void
Dijkstra( int StartVertex, const Graph& G,
          PathTableEntry T[] )
{
   InitTable( StartVertex, G, T );
   int V;
   AdjVertex Nbr;

   for( ; ; )
   {
      V = MinimumUnknown( G, T );
      if( V == -1 )  break;

      T[V].PathKnown = true;
      for each Nbr in V.Edges()  {
         if( !T[Nbr.W].PathKnown &&
             T[V].Distance + Nbr.Weight < T[Nbr.W].Distance )
         {
            // Update TableEntry for Nbr.W
            DecreaseWeight( Nbr.W, V, T[V].Distance + Nbr.Weight );
            T[Nbr.W].PathPred = V;
         }
      }
   }
}
```

- The functions `MinimumUnknown` and `DecreaseWeight` and the method for extracting an actual path once the algorithm is completed will be discussed in class.
- After working through the exercises, we will prove that the algorithm is correct.

Exercises

1. Simulate Dijkstra's algorithm on the following graph. (Assume 2 is the start vertex.) Each edge is specified by two numbers in the adjacency list: the adjacent vertex `W`, and the `Weight` (distance or cost) of the edge. These are the two member variables in `struct AdjVertex`. Thus, for example, vertex 3 has two outgoing edges: `(1,4)` goes to vertex 1 with a weight of 4, and `(4,8)` goes to vertex 4 with a weight of 8.

   ```
   1: (2,2), (3,2), (4,1), (5,7)
   2: (1,3), (3,1), (4,5)
   3: (1,4), (4,8)
   4: (5,2)
   5: (3,9)
   ```

2. The most important part of the algorithm left unspecified is the function `MinimumUnknown( G, T )` in the line

   ```
   V = MinimumUnknown( G, T );
   ```

   As discussed in class, this function should find the vertex having the smallest distance among the vertices with `!PathKnown`. There are several ways to implement `MinimumUnknown( G, T )`:

   (a) As simply a linear search through `T` to find the vertex with the smallest distance having `PathKnown == false`. What is the overall worst case complexity of the algorithm in this case?

   (b) Using a priority queue (or a binary heap) to store the `Distance` of the vertices with `PathKnown == false`.

      i. What crucial operations are needed on the binary heap data structure to implement this?

      ii. Which should occur in `MinimumUnknown` and which occur in `DecreaseWeight`?

      iii. What is the overall worst case complexity of the algorithm when using a priority queue?

3. Dijkstra's algorithm can be made more efficient in two special cases: when the edge distances are all the same, and when the graph is acyclic. Outline modifications to the algorithm for each of these cases and find the worst-case complexity of the resulting algorithm in each case.

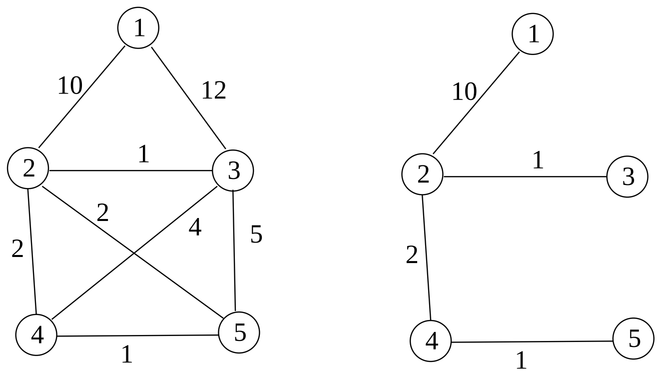
4. In a C++ implementation, Dijkstra's algorithm could be a class object. This object would accept a graph and a start vertex as input to its constructor, it should run the actual algorithm, and then it should answer questions about the results, such as the shortest path to a particular vertex and the length of certain paths.

   (a) Outline the private structures and member variables this class should define.

   (b) Outline the public and private member functions this class should define.

Minimum Spanning Trees — Prim's Algorithm

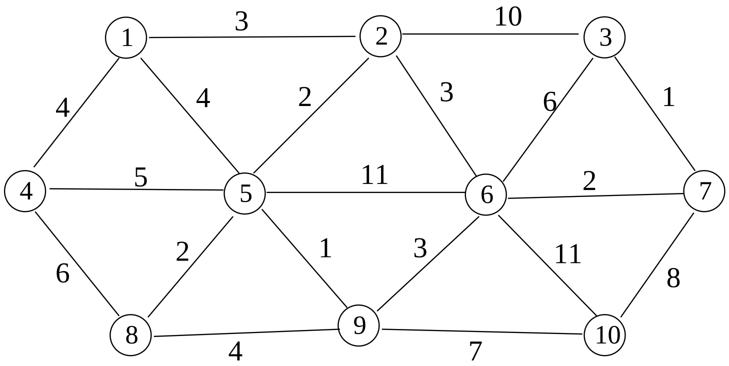
- **Definition:** Given an undirected graph, $G = (V, E)$, a *spanning tree* $T = (V_T, E_T)$ is a subgraph of G such that $V_T = V$ and T is a tree.
- **Definition:** If there are costs (weights or distances) associated with the edges in G, a *minimim spanning tree* (MST) is a spanning tree where the sum of the weights in E_T is minimal.
  - In the following, the left shows a graph and the right shows one of its minimum spanning trees.



  - Minimum spanning trees are not necessarily unique.
- There are two algorithms in the text, one is Prim's and the other is Kruskal's. You are only responsible for Prim's algorithm.
- Here is an outline of Prim's algorithm:
  1. Starting from an empty tree, T, pick a vertex, v_0, at random and initialize: $V_T = \{v_0\}$ and $E_T = \{\,\}$.
  2. Find a vertex, $v \in V - V_T$, such that v has the smallest weight edge (u, v) joining it to a vertex $u \in V_T$. In other words, of all edges (u', v') such that $u' \in V_T$ and $v' \in V - V_T$, none has a smaller weight than (u, v).
  3. Add v to V_T and (u, v) to E_T.
  4. Repeat until $V_T = V$.

Exercises

1. Use the outline of Prim's algorithm to find a MST in the following graph. Just mark the vertices and edges that will be in your MST.



2. Prim's algorithm can be implemented by making a few minor modifications to Dijkstra's. What are these modifications?
3. What is the complexity of Prim's algorithm? What is the complexity in the special case that all edges have the same weight?

Topological Sort

- **Definition:** A topological sort of the nodes of a directed graph, $G = (V, E)$, is an ordering of the vertices in V such that for any pair of distinct vertices, v_i and v_j, if there is a path in G from v_i to v_j then v_i must occur before v_j in the ordering.
- The algorithm is based on the notion of the "indegree" of a vertex. We will go over the idea with an example in class.

Here is a modified version of the author's pseudo-code for the algorithm. The code assumes there is a `Vertex` class object for each vertex which stores the `InDegree` and `TopNum`. The code also assumes the `InDegree` has been precomputed for each vertex.

```
void
Topsort( Graph& G )
{
    unsigned int counter = 0;
    Vertex v, w;
    queue<Vertex> Q;

    for each vertex v in G {
      if( v.InDegree == 0 )
          Q.push( v );
    }

    while( !Q.empty( ) ) {
      v = Q.front( );
      Q.pop();
      v.TopNum = ++counter;
      for each Vertex w adjacent to v {
          if( --w.InDegree == 0 )
              Q.push( w );
      }
    }
    if( counter < G.Num_Vertex() )
        Error( "Graph has a cycle" );
}
```

Exercises

1. Give the `TopNum` value assigned to the vertices in the following graphs. Each graph is represented by an adjacency list, where numbers appearing after a ":" indicate directed edges in the digraph. For example, `3: 2, 8` means there is a directed edge from vertex 3 to vertex 2 and a directed edge from vertex 3 to vertex 8.

   Graph 1

   ```
   1:
   2: 1
   3: 2, 8
   4: 5, 6
   5: 3, 6, 7
   6: 3
   7: 2, 1
   8: 1
   ```

   Graph 2

   ```
   1: 5
   2: 1
   3: 1, 2
   4: 1
   5: 2, 7
   6: 1, 3, 4
   7:
   ```

2. Suppose `Q` was a stack instead of a queue in `Topsort`. Would the algorithm still work? Do you prefer this solution?

3. What is the time complexity of `Topsort`? Justify your answer.

4. This exercise explores some implementation details.

   (a) Provide a declaration for the class `Vertex`.

   (b) `InDegree` and `TopNum` shouldn't be public member variables of class `Vertex`, yet the pseudo-code makes them appear to be. How should this apparent conflict be resolved?

   (c) In what type of data structure should these vertices be stored?

   (d) Write an efficient algorithm to compute `InDegree` for each vertex.

   (e) Does the `queue` need to store `Vertex` class objects? What is the cost of doing so? What else might it store?

Depth-First Search and Breadth-First Search

- Many graph algorithms depend on Depth-First Search (DFS) or Breadth-First Search (BFS) as a major step to compute graph properties and organize graph vertices for further computation.

- Here is the basic DFS algorithm, defined recursively. It just marks a vertex as "visited" by setting the value of a boolean class member variable. (An initialization step before the call to DFS sets `!V.Visited` for each vertex.) Algorithms built on DFS enhance the step of marking a vertex as visited with more substantial operations.

```
DFS( Vertex& V )
{
   V.Visited = true;
   for each W adjacent to V
      if ( ! W.Visited ) DFS(W);
}
```

  This algorithm requires $O(|V| + |E|)$ time.

- Here is the basic BFS algorithm. It is non-recursive.

```
BFS( Vertex& V )
{
   V.Visited = true;
   queue<Vertex> q;
   q.push( V );
   while ( !q.empty() ) {
      W = q.front();  q.pop();
      for each U adjacent to W
        if ( ! U.Visited ) {
           U.Visited = true;
           q.push( U );
        }
   }
}
```

  This algorithm also requires $O(|V| + |E|)$ time.

Exercises

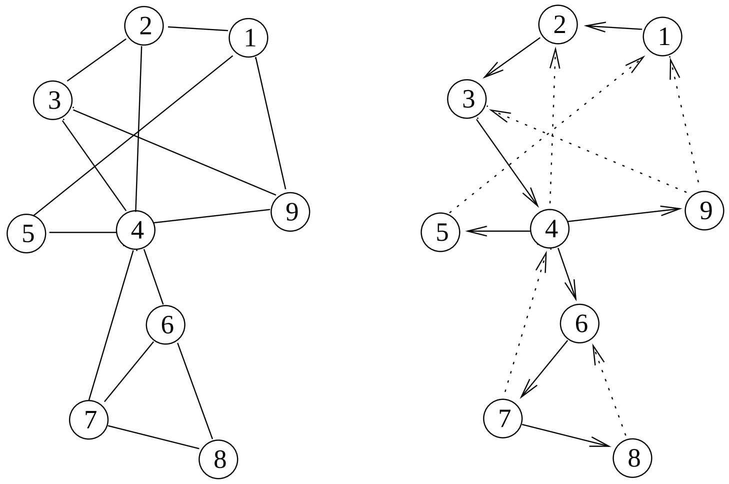
1. In BFS, why are `V` and `U` marked as visited before pushing them onto the queue instead of afterwards?

2. How can BFS be turned into a non-recursive version of DFS?

3. How can BFS or DFS be used to determine if an undirected graph, G, is connected and to find its connected components? In particular, write a DFS algorithm to label the connected components in G. Each vertex in a given connected component should have the same label (e.g. a positive integer), which is different from the labels of the vertices in other connected components. You will need to modify DFS and write an outer function that repeatedly calls DFS.

Finding Articulation Points

- **Definition:** An *articulation point* of a connected, undirected graph, G, is a vertex whose removal makes G disconnected.
- Designing the algorithm will require
  - defining *tree edges* and *back edges*
  - defining `v.Num` and `v.Low`, which can be implemented as member variables in the `Vertex` class or in a subclass of `Vertex` specific for use in the articulation point algorithm.
- Here is a graph numbered according to a depth-first traversal (left) and its edges redrawn (right) to distinguish tree edges and back edges. It will be used in class to describe the articulation points algorithm.



- Here is the pseudo-code for the recursive version of the algorithm. It is assumed that the original graph is connected (checking is trivial) and that the algorithm returns a list of articulation points:

```
// Counter is global and initialized to 1.
// Parent is the parent of a vertex in the DFS tree
void
Find_Art( Vertex & V, list<Vertex> & art_points )
{
   V.Visited = true;
   V.Low = V.Num = Count++;     // Rule 1.
   for each vertex W adjacent to V
      if( ! W.Visited )            // Forward edge.
      {
         W.Parent = V;
         Find_Art( W, art_points );
         if( W.Low >= V.Num )  art_points.push(V);
         V.Low = Min( V.Low, W.Low );     // Rule 3.
      }
      else if( V.Parent != W )            // Back edge.
         V.Low = Min( V.Low, W.Num );     // Rule 2.

 }
```

Exercises

1. What is the output of the `Find_Art` for the following graph? Assume `A` is the first node visited. Also, show the values of `Num` and `Low` for each node of the graph. Note that letters are used as vertex labels instead of numbers to help avoid confusion in hand-simulating the algorithm.

```
A: B, C
B: A, D
C: A, D, E, F
D: B, C
E: C, I
F: C, G, H
G: F, H
H: F, G
I: E
```

2. Actually, the `Find_Art` algorithm is not completely correct, as you might have discovered: it does not handle the root of the DFS tree properly. Augment the code to correctly decide if the root is an articulation point.

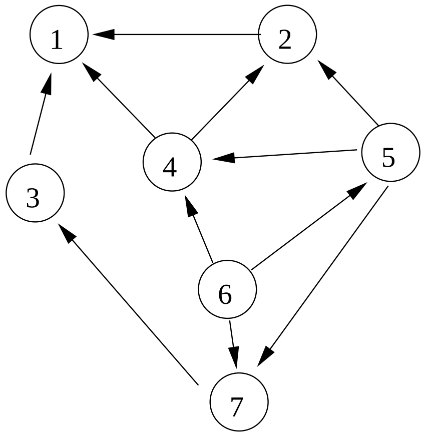
Review Problems:

This is a short set of problems which is not comprehensive. Rework the problems in the notes for more complete review.

1. Dijsktra's algorithm computes the shortest length (minimum cost) path from a starting vertex to each of the other vertices in a graph. There may, however, be more than one shortest path. Explain in words, not in pseudocode, how to modify Dijkstra's algorithm to produce a count of the number of shortest length paths from the start vertex to each of the other vertices.

2. Dijsktra's algorithm computes the shortest length (minimum cost) path from a starting vertex to each of the other vertices in a graph. There may be, however, more than one shortest path. Suppose that among these equal distance shortest paths, you wanted the algorithm to pick the path having the fewest edges. Explain carefully in words, not in pseudocode, how to modify Dijkstra's algorithm to do this.

3. (a) Find a topological sort of the following directed acyclic graph. You need only list the vertices in order, although showing your work will help earn partial credit if you make a mistake.



   (b) This graph has more than one topological sort. Briefly describe the general structure of directed acyclic graphs that have exactly one topological sort.