Chapter 8: Graph Algorithms

Contents

8.1 Introduction to graphs
   8.1.1 Graph representation
   8.1.2 Traversals
   8.1.3 Topological sorting
   8.1.4 Connectedness
8.2 The Minimum spanning tree
   8.2.1 Prim’s algorithm
   8.2.2 Kruskal’s algorithm
8.3 Disjoint sets
   8.3.1 Equivalence relation
   8.3.2 Kruskal’s Algorithm
   8.3.3 Maze generation
   8.3.4 Union-Find
   8.3.5 The Ackermann function
8.4 The Shortest path problems
   8.4.1 Dijkstra’s algorithm
   8.4.2 Graphs with negative costs
   8.4.3 All-pairs shortest paths

8.1 Introduction to Graphs

A. Aho and J. Ulman acknowledge that “Fundamentally, computer science is a science of abstraction.” Computer scientists must create abstractions of real-world problems that can be represented and manipulated in a computer.

Graphs are used to model a set of connections between pairs of items. Consider scheduling final exams. For successful scheduling we have to take into account associations between courses, students and rooms. In our model the set of items (courses, students and rooms) won't be very helpful. We also have to have a set of connections between pairs of items, because we need to study the relationships between connections.

The basic idea of graphs was introduced in the 18th century by the great Swiss mathematician Leonhard Euler. He used graphs to solve the famous Königsberg bridge problem.
The German city of Königsberg (now it is Russian Kaliningrad) was situated on the river Pregel. It had a park situated on the banks of the river and two islands. The mainland and the islands were joined by seven bridges. The problem was whether it was possible to take a walk through the town in such a way as to cross over every bridge once, and only once. Here is the graph model of the problem

![Graph Model of the Königsberg Problem]

**Definition.** A graph is a set of points (we call them vertices or nodes) connected by lines (edges or arcs). The simplest example known to you is a linked list.

Throughout this chapter we'll use the following notations

\[ G = (V, E) \]

- \( V \) is a graph with \( V \) vertices and \( E \) edges
- \( |V| \) is the number of vertices
- \( |E| \) is the number of edges

A substantial amount of definitions are associated with graphs. A graph that has nonempty set of vertices connected at most by one edge is called **simple**. **Multigraphs** allow multiple edges between two vertices. **Pseudographs** allow edges to connect a vertex to itself.
What do these three types of graphs have in common? The set of edges is \textit{unordered}. All such graphs are called \textit{undirected}. A \textbf{directed} graph consist of vertices and ordered pairs of edges. Note, multiple edges in the same direction are not allowed.

If multiple edges in the same direction are allowed, then a graph is called \textbf{directed multigraph}.

Usually by a graph people mean a simple undirected graph. No directions, no self-loops, no multiple edges. But be careful and watch out!

An edge may also have a \textbf{weight} or \textbf{cost} associated with it. If \((a, b)\) is an edge we might denote the cost by \(c(a, b)\) In the example below, \(c(a, b) = c(b, a) = 7\).

Two vertices are called \textbf{adjacent} if there is an edge between them. The \textbf{degree of a vertex} in an undirected graph is the number of edges associated with it. If a vertex has a loop, it contributes twice.
In the above picture, the degree of vertex c is 4, and the degree of vertex b is 3.

**Theorem** (The handshaking theorem).

Let $G$ be an undirected graph (or multigraph) with $V$ vertices and $E$ edges. Then

$$2E = \sum_{v=V} \deg(v)$$

Example,

$$\deg(a) = 4 \quad \deg(b) = 3 \quad \deg(c) = 3 \quad \deg(d) = 4$$

$$2 \times 7 = 4 + 3 + 3 + 4$$

**Exercise.** Suppose a simple graph has 15 edges, 3 vertices of degree 4, and all others of degree 3. How many vertices does the graph have?

In a directed graph terminology reflects the fact that each edge has a direction. In a directed graph vertex $u$ is adjacent to $v$, if there is an edge leaving $u$ and coming to $v$. In a directed graph the **in-degree** of a vertex denotes the number of edges coming to this vertex. The **out-degree** of a vertex is the number of edges leaving the vertex. A loop contributes 1 to both the in-degree and out-degree.

$$\text{in-deg}(0) = 2 \quad \text{in-deg}(3) = 3$$
in-deg(1) = 3  \quad in-deg(4) = 2
in-deg(2) = 2

**Theorem.** Let $G$ be a directed graph (or multigraph) with $V$ vertices and $E$ edges. Then

$$E = \sum_{v \in V} \text{indeg}(v) = \sum_{v \in V} \text{outdeg}(v)$$

A path is a sequence of distinctive vertices connected by edges. A path is simple if it doesn’t contain the same edge more than once. A cycle is a sequence of distinctive vertices that begins and ends at the same vertex.

Vertex $v$ is reachable from $u$ if there is a path from $u$ to $v$. A graph is connected, if there is a path between any two vertices. Otherwise, a graph is called disconnected. For directed graphs the situation is more complex. A directed graph is strongly connected if there is a path from $u$ to $v$ and from $v$ to $u$ for any $u$ and $v$ in the graph. A directed graph is weakly connected if the underlying undirected graph is connected.

**Exercise.** Given a graph with 7 vertices; 3 of them of degree two and 4 of degree one. Is this graph connected?

**Theorem.** In an undirected simple graph with $V$ vertices, there are at most $\frac{V(V-1)}{2}$ edges.

**Proof.** Consider an undirected graph with $V$ vertices and choose any vertex. The possible number of edges leaving this vertex is $V - 1$. Take another vertex. The possible number of edges leaving this vertex is $V - 2$ (don’t count the edge between these vertices twice!), and so on. We have

$$(V - 1) + (V - 2) + \ldots + 2 + 1 = \frac{V(V - 1)}{2}$$

**Exercise.** What is the maximum number of edges in a simple disconnected graph with $V$ vertices?

**Trees**

Let us consider another relation between the number of vertices and edges

- If graph is disconnected, then $E < V - 1$
- If graph is connected, then $E > V - 1$
- If graph is a tree, then $E = V - 1$

A tree is a connected graph without cycles. In other word, adding any new edge to a graph will form a cycle. A rooted tree is a tree with one vertex designated as a root.
A forest is a graph without cycles, but not necessarily connected. In other words, a forest is a set of trees.

A binary tree is a tree where each vertex (except a root) has degree either 1, 2 or 3. A leaf is a vertex of degree 1. An internal node is a vertex of degree 2 or 3.

A spanning tree of a graph is a subgraph, which is a tree and contains all vertices of the graph. In the figure below, the right picture represents a spanning tree for the graph on the left. A spanning tree is not unique.

A minimum spanning tree (MST) is a spanning tree of minimum total cost (the cost of the edges selected for the tree).

**Question.** Does every graph have a spanning tree?

### Famous Problems on Graphs

*The Euler cycle (or tour) problem:* Is it possible to traverse each of the edges of a graph exactly once, starting and ending at the same vertex?

*The Hamiltonian cycle problem:* Is it possible to traverse each of the vertices of a graph exactly once, starting and ending at the same vertex?

*The travelling salesman problem:* Find the shortest path in a graph that visits each vertex at least once, starting and ending at the same vertex?

*The planar graph:* Is it possible to draw the edges of a graph in such a way that edges do not cross?

*The four coloring problem:* Is it possible to color the vertices of a graph with at most 4 colors such that adjacent vertices get different color?

*The marriage problem (or bipartite perfect matching):* At what condition will a set of boys be married to a set of girls such that each boy gets a girl he likes?

### 8.1.1 Graph Representation

Given a graph $G = (V, E)$. We have shown that for all graphs, the number of edges $E$ and vertices $V$ satisfies the following inequality

$$E < V^2$$

If the number of edges $E > V \ast \log V$, we say that this is a dense graph, it has a large number of edges. Otherwise, this is a sparse graph $E < V \ast \log V$. In most cases, the graph is relatively sparse.
There are two standard ways to represent a graph:
· as a collection of adjacency lists
· or as an adjacency matrix

The adjacency-list representation of a graph $G$ consists of an array of linked lists, one for each vertex. Each such list contains all vertices adjacent from a chosen one. Here is an adjacency-list representation:

Vertices in an adjacency list are stored in an arbitrary order. A potential disadvantage of the adjacency-list representation is that there is no quicker way to determine if there is an edge between two given vertices.

This disadvantage is eliminated by an adjacency matrix representation. The adjacency matrix is a matrix of size $V \times V$ such that

$$M_{ij} = \begin{cases} 1, & \text{if there is an edge between } i \text{ and } j \\ 0, & \text{otherwise} \end{cases}$$

and for a weighted graph

$$M_{ij} = \begin{cases} c(i, j), & \text{if there is an edge between } i \text{ and } j \\ \infty, & \text{otherwise} \end{cases}$$

An adjacency list representation is used for representation of the sparse graphs. An adjacency matrix representation may be preferred when the graph is dense. The space used in the adjacency list representation is $O(V + E)$. The matrix uses $O(V^2)$. 
8.1.2 Graph Traversal

Graph traversal means visiting all vertices in a systematic order. We can choose any vertex as a starting point. Then we will systematically enumerate all vertices accessible from it. Because a graph might contain cycles, we need some way for marking a vertex as having been visited. To do so we will have a boolean array \texttt{visited} with all elements initially set to false. We will set a correspondent element to true as soon as we visit a particular vertex. Also, we need to keep in mind that the graph might be disconnected.

There are two of the most common traversals:

- depth-first search (DFS) and
- breadth-first search (BFS)

DFS uses a \texttt{stack} for backtracking. BFS uses a \texttt{queue} for bookkeeping.

On each step of DFS you go deeper and deeper into the graph. On each step of BFS you first visit all siblings and then their children. In case of BFS, we will use a queue as an intermediate data structure and for DFS we will use a stack. Depth-first search in an undirected graph is analogous to wandering in a labyrinth with a string and a can of paint.

Here is implementation of DFS (in pseudo-code):

\begin{verbatim}
for k := 1 to |V| do 1 visited[k] := false;
Create(Stack);
visited[startNode] := true;
Push(startNode);
while not IsEmpty(Stack) do
    nextNode := Pop(Stack);
    Print(nextNode);
    for each node adjacent to nextNode do
        if !visited[node] then
            visited[node] := true;
            Push(node);
\end{verbatim}

The worst-case running time of the traversal algorithm is \(O(V + E)\), if we use an adjacency list.

Exercise. Perform DFS on the following graph
The result of traversal gives the following spanning tree:

Note, the result of traversing is not unique.

**Exercise.** Perform BFS on the above graph

**Exercise.** What is the worst-case running time of the BFS algorithm if adjacency matrix is used?

- **Applications of DFS**
  
The algorithm makes the foundation for algorithms for solving many problems on graphs
  - Determine the connected components of a graph
  - Find a cycle in a graph
  - Determine if a graph is bipartite.
  - Topologically sort a directed graph
8.1.3 Topological sorting

Suppose each vertex represents a task that must be completed, and an edge \((u, v)\) indicates that task \(u\) depends on task \(v\). That is \(v\) must be completed before \(u\). The topological ordering of the vertices is a valid order in which you can complete the tasks.

If \(G\) is a DAG (direct acyclic graph), then there exists an order in which the vertices can be listed such that if vertex \(u\) is reachable from vertex \(v\) (there is a path from \(v\) to \(u\)) then \(u\) is listed after \(v\). This is called "topological order".

It should also be evident from this picture that a topological sort is not unique. For example, the following are valid topological sorts:

- \{a, b, c, d, e, f, g, h, i\}
- \{a, c, b, f, e, d, h, g, i\}
- \{a, c, f, b, e, h, d, g, i\}

In other words, we want to arrange the vertices along a line so that all edges go from left to right:

Algorithm.

It's easy to see that such an ordering exists. One way to find a topological sort is to consider the in-degrees of the vertices. Clearly the first vertex in a topological sort must have in-degree zero and every DAG must contain at least one vertex with in-degree zero.

- Select a vertex that has in-degree zero.
• Add the vertex to the sort.
• Delete the vertex and all its edges from the graph.

**Exercise.** List the order in which the nodes of the directed graph shown below are visited by a topological order traversal that starts from vertex a.

![Directed Graph](image)

{a, c, g, d, b, e, f, h}

Another way is to find a vertex with no out-edges. Print it, delete it from the graph, and repeat. There is an algorithm based on DFS:

• call DFS and return vertex that has no undiscovered leaving edges
• insert it onto the front of a linked list
• return the linked list of vertices

In the above picture: start with $a \rightarrow c \rightarrow f \rightarrow h$, then backtrack from $h$ to $f$, making $h$ the first vertex with no undiscovered leaving edges. From $f$ backtrack to $c$, this makes $f$ another vertex with no undiscovered leaving edges. We put $f$ in the front of the linked list

$$f-h$$

From $c$ we could go to $d$, then to $e$. Backtrack to $d$

$$e-f-h$$

From $d$ go to $b$. Backtrack to $d$

$$b-e-f-h$$

Backtrack to $c$

$$d-b-e-f-h$$

From $c$ we can go $g$. Backtrack to $c$

$$g-d-b-e-f-h$$

$$c-g-d-b-e-f-h$$

$$a-c-g-d-b-e-f-h$$

**IMPLEMENTATION NOTES:**
In order to implement the algorithm, an array of integers is used to record the in-
degrees of the vertices. On each iteration, we decrease in-degrees of all the successors of the vertex. In addition, we use a queue to keep track of the vertices that have not yet been visited, but whose in-degree is zero. When the in-degree of a vertex becomes zero, that vertex is enqueued.

The worst-case running time of the topological sort is $O(V + E)$.

### 8.1.4 Connectedness

#### Connectedness of an Undirected Graph

A traversal of an undirected graph (either depth-first or breadth-first) starting from any vertex will only visit all the other vertices of the graph if that graph is connected. Therefore, there is a very simply way to test whether an undirected graph is connected. Count the number of vertices visited during a traversal of the graph. Only if all the vertices are visited is the graph connected.

#### Connectedness of a Directed Graph

When dealing with directed graphs, we define two kinds of connectedness, strong and weak.

The subgraph \{a, b, c\} below is strongly connected, however the whole graph is not (there is no path from d to b)

However, the graph "looks" connected. This is represented by a weak connection. A weakly connected directed graph corresponds to the underlying undirected connected graph.

**Algorithm.**

A simple way to test whether a directed graph is strongly connected uses V traversals - one starting from each vertex. Each time the number of vertices visited is counted. The graph is strongly connected if all the vertices are visited in each traversal. The worst-case running time of this algorithm is $O((V + E) \times V)$.

There's a very efficient algorithm due to Tarjan for computing the strong components of a graph in $O(V + E)$ time using a depth first search. The proof that it works is beyond the scope of this course.

```c
Main (G)
{
    int k = 0;
```
for all x in V do num[x] = 0;
for all x in V if (num[x] == 0) STRONG (v);
}

STRONG (v) {
    k++;
    num[v] = k;
    lowlink[v] = k;
    push v onto stack S;
    for all w in adj (v) do
    {
        if (num[w] == 0)
        {
            STRONG (w);
            lowlink[v] = min (lowlink[v], lowlink[w]);
        }
        else if (num[w] < num[v])
        {
            if w is in S then lowlink[v] = min (lowlink[v], num[w]);
        }
    }
    if (lowlink[v] == num[v])
    {
        pop vertices off the stack while (num[stack top] >= num[v])
        output all of these vertices as a strong component
    }
}

- Finding a cycle in a directed graph
  As the analysis of topological sort shows, all we really need to do this is to keep track of which vertices are on the stack. We can do this explicitly by adding a new field for each vertex called mark(v).

- Special Cycles
  Definition. An Euler cycle (or circuit) is a cycle that traverses every edge of a graph exactly once. If there is an open path that traverse each edge only once, it is called an Euler path.

  Exercise. Does the graph below have an Euler cycle?
Definition. A Hamiltonian cycle is a cycle that traverses every vertex of a graph exactly once. If there is an open path that traverses each vertex only once, it is called a Hamiltonian path.

Exercise. Does the graph below have a Hamiltonian cycle? (This is the Icosian puzzle, invented by Hamilton in 1857)

8.2 The Minimum Spanning Tree

The MST problem is one of simplest and best-studied optimization problems in computer science. Given undirected, connected graph with $E$ weighted edges. The problem is to find a spanning tree of minimum total weight.

Spanning tree is a connected and acyclic subgraph containing all the vertices. Minimum spanning tree (MST) is a spanning tree of a weighted graph with minimum total edge weight.

The weight of a spanning tree is the sum of the weights on all the edges which comprise the spanning tree

$$\sum_{(u,v)} c(u, v) \rightarrow \min$$
Consider Fred Hacker’s algorithm: using BFS, find ALL spanning trees and then pick one with the minimum cost. What’s wrong with this idea?

Cayley's Theorem. The number of spanning trees in complete graph $K_n$ with $n$ vertices is $n^{n-2}$.

The complete graph on $n$ vertices, denoted $K_n$, is a simple graph in which there is an edge between every pair of distinct vertices. Here are $K_4$ and $K_5$:

Exercise. How many edges in $K_n$?

8.2.1 Prim's Algorithm (1957)

The algorithm constructs the minimum-cost spanning tree of a graph by adding one vertex at a time. The algorithm constructs a sequence of trees, each of which is a subgraph. The algorithm begins with a tree that contains one selected vertex, say $A$. On each step, we choose a minimum-weight edge, connecting a vertex $v$ in set $A$ to a vertex $u$ outside of set $A$ and then adding $u$ to the tree
The implementation of the algorithm is based on the priority queue (heap) in which we keep distances to adjacent vertices. On each step we deleteMin from the queue and update the queue to reflect new distances.

The worst-case running time is $O(V \log V + E \log V)$ when adjacency list is used.

- deleteMin takes $O(\log V)$ and do this $V$ times
- fix a heap using percolation takes $O(\log V)$ but might do it for all edges

The time taken by Prim's algorithm is determined by the speed of the heap operations. With the queue implemented as a Fibonacci heap, it takes $O(E + V \log V)$ time.

### 8.2.2 Kruskal's Algorithm (1956)

The algorithm constructs the minimum-cost spanning tree of a graph one EDGE at a time. Initially, each vertex is in its own tree in the forest. Then, the algorithm considers each edge in turn, order by increasing weight and chooses the cheapest edge. If an edge $(u, v)$ connects two different trees, then $(u, v)$ is added to the set of edges of the MST, and two trees are merged. On the other hand, if an edge $(u, v)$ connects two vertices in the same tree, then the edge is discarded.
If a priority queue (heap) is used to organize the edges, it takes
\[ O(\log E) \] time to delete a shortest edge while maintaining the PQ
\[ O(E \log E) \] time to delete all edges
\[ O(V) \] time to join two clusters.
\[ O(E \times V) \] time to join all clusters.

Totally, Kruskal's algorithm takes \( O(E \times V + E \times \log E) \) time.

In the next chapter we will discuss a disjoint sets data structure, that will speed up the process of joining clusters to amortized constant time.
**Correctness**

The proof of correctness is based on the following

**Lemma:** Let $X$ be any subset of the vertices of $G$, and let edge $e$ be the smallest edge connecting $X$ to $G-X$. Then $e$ is part of the minimum spanning tree.

**Proof.** By contradiction. Suppose a tree $T$ is the MST and does not contain the smallest edge $e$. Denote $e = (u, v)$, where $u$ in $X$ and $v$ not in $G-X$. Since $T$ is a spanning tree, there is a unique path from $u$ to $v$. This path together with edge $e$ forms a cycle in $G$. This path has to include another edge $e_1$ connecting $X$ to $G-X$. Consider another spanning tree $T_1$ which is created by removing $e_1$ and adding $e$, namely $T_1 = T - e_1 + e$. Tree $T_1$ has a smaller cost, since $e < e_1$. Thus, $T_1 < T$. Contradiction

### 8.3 Disjoint Sets

#### 8.3.1 Equivalence Relation

Disjoint sets are called equivalence classes. Each class consist of elements that are related to each other according to some rule $R$, that is in turn called an equivalence relation.

Suppose you have a group of people. We want to figure out who is related to whom. Our relations are in the form "Alice is related to Bob".

Surely, "Alice is related to Alice".

But what about "Bob is related to Alice"? Is it true?

Now suppose that "Bob is related to Trudy".

Is it true that "Alice is related to Trudy"?

**Definition.** A binary relation $R$ over a set $S$ is called an equivalence relation if it has the following property

1. **Reflexivity.** $\forall x, \ x R x$
2. **Symmetry.** $\forall x, y \ x R y \iff y R x$
3. **Transitivity.** $\forall x, y, z, \ x R y \text{ and } y R z \implies x R z$

**Theorem.** Let $R$ be an equivalence relation over a set $S$. Then $S$ can be written as a union of disjoint sets

$$S = \bigcup_k S_k$$

with the following property

1. if $x \in S$ and $y \in S$ then $x R y$ if and only if $x$ and $y$ belong to the same $S_k$
2. Subsets $S_k$ are non-empty and pair-wise disjoint.

Such sets $S_k$ are called equivalence classes.
Example. Consider set $\mathbb{Z}$ of all integers and define a relation $R$ by saying that $x$ and $y$ are related if $x - y$ is divisible by 2. Is this an equivalence relation?

Answer:
Reflexivity. $x - x = 0$, which is divisible by 2
Symmetry. if $x - y$ divisible by 2 then $y - x = -(x - y)$ divisible by 2
Transitivity. $x - y = 2m$, $y - z = 2n$ then $x - z = (x - y) - (z - y) = 2(m - n)$

Example.
What equivalence classes does the above relation define?
Given that $x - y = 2m$. Let $y = 2k$, it follows that $x$ must be even as well. Let $y = 2k + 1$, then $x$ must be odd. Therefore, there are two equivalence classes, even and odd numbers.

Exercise. Define an equivalence relation on graphs?

Exercise. What equivalence classes does the above relation define?

### 8.3.2 Kruskal’s Algorithm

The algorithm constructs the minimum-cost spanning tree of a weighted graph. We use the disjoint set data structure. Initially, we put each vertex into a separate set (an array). The algorithm considers each edge in order by increasing weight. An edge $(u, v)$ is added to the set of edges of the MST, if vertices $u$ and $v$ belong two different sets. These two sets (arrays) of vertices are merged. Otherwise, the edge is discarded.

We will demonstrate the algorithm on the following graph.

![Diagram of graph]

This diagram traces the state of disjoint sets after each iteration of the algorithm.
If a priority queue (heap) is used to organize the edges, it takes
\( O(\log E) \) time to delete a shortest edge while maintaining the PQ
\( O(E \log E) \) time to delete all edges
\( O(V) \) time to join two clusters.
\( O(E \times V) \) time to join all clusters.
Totally, Kruskal's algorithm takes \( O(E \times V + E \times \log E) \) time.

### 8.3.3 Maze generation

A good way to describe mazes is to ignore the geometry and focus on topology. A maze is basically a graph: vertices are rooms and edges are corridors. Properties of standard mazes are
- **random** - walls to remove should be selected randomly
- **solvable** - there should be a single path between the start cell and the end cell
- **reachable** - there are must be no rooms that are completely blocked off from the rest of the maze

To generate a maze you basically runs Kruskal's algorithm. Start with a rectangular grid of squares (each room is isolated)

```
0 1 2 3 4 5
6 7 8 9 10 11
12 13 14 15 16 17
18 19 20 21 22 23
```

Randomly knock down some walls (lines in the grid) to find a path from the start (cell
0) to the end (cell 23).

The difficulty in generating a good maze is in wall removal. We shall use the disjoint set data structure. Initially, we put each room into a separate set. When we remove a wall, we join two sets together, however we have to check in advance whether two rooms (on either side of the wall) belong to different sets. Rooms that are connected by some path are in the same set. We stop when only one set left.

Here is a maze generation algorithm

```java
create a data structure called S
for(int i = 0; i < 24; i++)  S.makeset(i);
while(S.find(0) != S.find(23))
{
    choose randomly a pair (i, j) of adjacent cells;
    if(S.find(i) != S.find(j))
    {
        knock down the wall;
        S.union(i, j);
    }
}
```

### 8.3.4 Union-Find

Both above applications use a structure for maintaining a collection of disjoint sets. This data structure is supposed to support the following operations

- MAKESET(x) - creates a new set containing a single element x.
- UNION(x, y) - joins two sets containing x and y together
- FIND(x) - returns the name of the set containing x.

We will consider how to implement those two applications efficiently. We measure the cost of UNIONs and FINDs operations on n initial sets.

**Array Implementation**

The simplest implementation is to represent each set as an array
FIND has cost $O(\text{length of the list})$
UNION has cost $O(\text{length of the list})$

- **Linked List Implementation**
  Represent each set as a linked list (where we keep track of both the head and the tail).
  FIND has cost $O(\text{length of the list})$
  UNION has cost $O(1)$

- **Forest Data Structure**
  We maintain a collection of trees. Searching in a tree has a log time! Initially, there are $n$ trees each containing a single node.
  FIND($i$) returns the root of a tree containing $i$
  UNION($i$, $j$) merges two trees containing $i$ and $j$
  Each tree is organized in such a way that children point to their parent. Sometimes it is called as a reversed tree. Here is an example, FIND(4), we start with 4 going up to the root 5

![Diagram of a forest data structure with FIND(4) example]

UNION($x$, $p$) joins two trees by hanging the root of one tree to the root of the other

- **Implementation**
  The basic ideas is not to actually maintain a tree data structure, but rather simulate it by a simple array. We store parent pointers in an array $\text{parent}[k]$. If $k$ is the root then its value is $k$. 
**initialization:**
for(int k = 0; i < n; i++)
    parent[k] = -1;

**find(k):**
for(j = k; parent[j] >= 0 && j != parent[j]; j = parent[j]);
return j;

**union(i, j):**
root1 = find(i);
root2 = find(j);
if(root1 != root2)
    parent[root2] = root1;

**Example**

```
0 1 2 3 4 5 6 7 8
-1 -1 -1 -1 -1 -1 -1 -1
```

UNION(2, 3)

```
0 1 2 3 4 5 6 7 8
-1 -1 3 3 -1 -1 -1 -1
```

UNION(6, 8)

```
0 1 2 3 4 5 6 7 8
-1 -1 3 3 -1 -1 8 -1 8
```

UNION(0, 2)

```
0 1 2 3 4 5 6 7 8
3 -1 3 3 -1 -1 8 13
```

UNION(2, 6)
- **Worst-Case Complexity**

FIND has cost $O(n)$, the tree can be degenerated to a linked list.

UNION has cost $O(1)$

To improve the performance of this data structure we will be using the following two heuristics.

**Union by Rank**

Maintain the height of all nodes. During UNION, make a shorter tree a subset of a taller tree. If two roots have the same rank, union them in any order and then increase the rank of the new root by 1.

A bit of improvement - FIND has cost $O(\log n)$ in the worst case.

**Path Compression**

Since, anyway, we travel the path to the root during a FIND operation, we might as well redirect all nodes on the path directly to the root. The picture below demonstrates a call to FIND($z$)

Note, during the path compression, we do not change ranks. In other words, only the root rank can be changed. Thus, ranks are monotonically increasing in the tree, as we traverse from a node to the root of the tree.

**Theorem.** A sequence of $N$ union and find operations of $N$ elements takes $O(N \log^* N)$ time.

The proof will be given in 15-451. Here $\log^* x$ (log-star) is the number of times we need to apply log to get 1. For example,

$$\log^* 16 = 3 \quad \text{since} \quad \log \log \log 16 = 3$$

$\log^* x$ is the inverse of the following function

$$A(0) = 1 \quad \text{and} \quad A(n) = 2^{A(n-1)}$$

which is a particular case of more general function known as the Ackermann function (see case 3 in the next section)
8.3.5 The Ackermann function

The Ackermann function is defined by
\[
A(0, n) = n + 1 \\
A(m + 1, 0) = A(m, 1) \\
A(m + 1, n + 1) = A(m, A(m + 1, n))
\]

It that grows faster that any primitive recursive function. Let us unwind a few particular cases.

Case \(m = 0\):
\[
A(1, 0) = 2 \\
A(1, n + 1) = A(1, n) + 1
\]
which has solution \(A(1, n) = n + 2\).

Case \(m = 1\):
\[
A(2, 0) = 3 \\
A(2, n + 1) = A(2, n) + 2
\]
which has solution \(A(2, n) = 2n + 3\).

Case \(m = 2\):
\[
A(3, 0) = 5 \\
A(3, n + 1) = 2A(3, n) + 3
\]
which has solution \(A(3, n) = 2^{n+3} - 3\).

Case \(m = 3\) (related to \(\log^*\) function)
\[
A(4, 0) = 13 \\
A(4, n + 1) = 2^{A(4, n) + 3} - 3
\]
which has solution \(A(4, n) = 2^{2^{\cdot \cdot \cdot}} - 3\) (exponentiation is \(n + 3\) times). To see how big that number, we compute it for \(n = 2\),
\[
A(4, 2) = 2^{2^{2^{\cdot \cdot \cdot}}} - 3 = 2^{65,536} - 3
\]
Regardless of its astronomical growth, Ackermann's function is Turing computable.
8.4 The Shortest Path Problems

8.4.1 Dijkstra's Algorithm

Consider a directed or undirected graph where each edge has a nonnegative weight. One of the nodes is designated as a source $s$. The problem is to find the shortest directed path between $s$ and ALL of the other vertices in the graph. By a shortest path we mean a set of edges with the minimum possible sum of their weights.

We will consider the following versions of the problem

- Unweighted Graph
- Directed Graph
- Single Source
- Negative weights
- All-pairs

Unweighted shortest path problem

Given an undirected unweighted graph $G$, and a source, find the shortest path (with the fewest edges) in $G$ from the source to all other vertices. We can adapt the BFS algorithm to this problem (the only change is that we maintain a distance field for each vertex in the graph):

```c
C={} ; //a list of visited vertices
Q.enqueue(s); //a queue
s.distance = 0;
while (!Q.isEmpty())
{
    v = Q.dequeue() ;
    for each w in adjacent(v) do
        if (w not in C)
        {
            Q.enqueue(w);
            C = C union w;
            w.distance = v.distance + 1;
        }
}
```

Graphs with positive integer weights

We can reduce this problem to the previous one. All that is required is to replace each edge of weight $W$ by $(W - 1)$ vertices of weight 1. Is this a practical approach? What's its complexity?

Single source shortest path

Find a shortest path from a given source to all other vertices.
Edsger Wybe Dijkstra (1930 - 2002)

Here are a few of his famous sayings

*The question of whether computers can think is like the question of whether submarines can swim.*

*Computer science is no more about computers than astronomy is about telescopes.*
Algorithm

The idea is similar to sorting. We need to order vertices \( s \), \( s_1 \), \( s_2 \), ..., \( s_n \) in such a way that \( d(s, s_1) \leq d(s, s_2) \leq ... \leq d(s, s_n) \), where \( d(s, s_k) \) is a shortest distance from the starting node \( s \) to \( s_k \). A shortest path from \( s \) to \( s_k \) may include any of \( \{s_1, s_2, ..., s_{k-1}\} \) but cannot include any of \( \{s_{k+1}, ..., s_n\} \).

The algorithm proceeds in a greedy fashion and divides all vertices into two groups
- vertices whose shortest path from the source is known
- vertices whose shortest path from the source is NOT known

At each step the algorithm, we move vertices one at a time from the unknown set to the known set. In the picture below, the known set (inside the cloud) is represented by vertices \( \{s, 1\} \) and the unknown set contains all other vertices.

![Graph Image]

The algorithm is almost identical to Prim’s algorithm. Prim's algorithm stores a minimum cost edge, but Dijkstra’s stores a minimum distance from the source to a vertex.

Implementation (using a priority queue)

Let \( D(v) \) denote a length from the source \( s \) to vertex \( v \).

**Initialization:** \( D(s) = 0 \)

For all \( v \neq s \), \( D(v) = \infty \)

Insert all vertices into a PQ, using \( D(v) \) as a key.

**Loop:**

Delete a node \( v \) from PQ using \( deleteMin() \)

Update \( D(w) \) for all \( w \) in \( \text{adj}(v) \) using \( decreaseKey() \)

\[
D(w) = \min\{D(w), D(v) + c(v, w)\}
\]

Updating is called relaxation. Note, each edge is relaxed only once. At each step the algorithm has examined only a part of the graph, and the current costs (array \( D \)) reflect the best cost. The table below reflects changes in the array \( D \) after each step of the algorithm.
Exercise. Perform Dijkstra's algorithm on the following graph, starting at A

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>s</td>
<td>9</td>
<td>14</td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>44</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>34</td>
<td>51</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>45</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>9</td>
<td>32</td>
<td>45</td>
<td>34</td>
<td>14</td>
<td>16</td>
</tr>
</tbody>
</table>

Complexity (using heaps)

Initialization:
- Initialization of $D(v)$ for all vertices $O(V)$
- Initialization of PQ $O(V)$

Loop:
- `deleteMin` $O(\log V)$
  - for all in PQ $O(V \log V)$
- Relaxation
  - a) update PQ $O(\log V)$
  - b) for all edges $O(E \log V)$

Therefore, the total complexity is $O(V \log V + E \log V)$. If instead of a priority queue, we use a Fibonacci heap, then a queue update can be done in constant time, and the running time would be $O(V \log V + E)$.

Question. Assume that an array is used instead of a priority queue. What would the algorithm's running time in this case?

Question. The algorithm computes the shortest length between two vertices. How
would you find vertices on the shortest path?

- **Running time of Dijkstra's Algorithm**
  
  o) unweighted graph
  
  the algorithm running time $O(V + E)$ - using a BFS

  1) PQ is a linear array
      
      findMin takes $O(V)$ - for one vertex
      
      total findMin takes $O(V^2)$ - for all vertices
      
      relaxation takes $O(1)$ - for one edge
      
      total edge adjustment $O(E)$
      
      the algorithm running time $O(V^2 + E)$

  2) PQ is a binary heap
      
      findMin takes $O(\log V)$ - for one vertex
      
      total findMin takes $O(V \log V)$ - for all vertices
      
      relaxation takes $O(\log V)$ - for one edge
      
      total edge adjustment $O(E \log V)$
      
      the algorithm running time $O(V \log V + E \log V)$

  3) PQ is a Fibonacci heap
      
      findMin takes $O(\log V)$ - for one vertex
      
      total findMin takes $O(V \log V)$ - for all vertices
      
      relaxation (decreaseKey) takes $O(1)$ - for one edge
      
      total edge adjustment $O(E)$
      
      the algorithm running time $O(V \log V + E)$

  Note, if a graph is dense we use an array representation to get a runtime complexity $O(V^2)$, if a graph is sparse we use a PQ - $O(V \log V)$.

**Exercise.** Assume that your graph is a directed acyclic graph. Design an algorithm that would perform better than Dijkstra's.

**Question.** How is Dijkstra's algorithm related to sorting?

- **Proof of Correctness**

  For each node $u \in S$, $\delta(u)$ is the length of the shortest $s$-$u$ path.

  Proof by induction on $|S|$.

  Base case: $|S| = 1$ is trivial.

  Inductive hypothesis: Assume true for $|S| = k \geq 1$.

  Let $v$ be next node added to $S$, and let $u$-$v$ be the chosen edge.

  The shortest $s$-$u$ path plus $(u, v)$ is an $s$-$v$ path of length $\delta(v)$.

  Consider any $s$-$v$ path $P$. We'll see that it's no shorter than $\delta(v)$.

  Let $x$-$y$ be the first edge in $P$ that leaves $S$, and let $P_1$ be the subpath to $x$. 
\[
\text{len}(P) \geq \text{len}(P_1) + \text{edge}(x, y) \geq \delta(x) + \text{edge}(x, y) = \delta(y) \geq \delta(v)
\]

The last inequality follows from the fact that the algorithm chooses \( v \) instead of \( y \).
### 8.4.2 Graphs with negative costs

Consider this example. If you run Dijkstra algorithm on a graph in the left picture, you find that a shortest distance from $S$ to $B$ to be 3, however there is a shorter path via vertex $X$:

How would we go about it?

**Idea 1**: increment each edge cost by the smallest negative cost and then apply Dijkstra's algorithm. In this case we will be solving a different minimization problem - we will penalize longer paths. In the following graph, the shortest distance between $S$ and $X$ is via vertices $B$ and $C$. However, after reweighting, the shortest distance is via vertex $A$:

**Idea 2**: use the dynamic programming.

**Exercise.** Design an algorithm that uses dynamic programming and estimate its running time.

### The Bellman-Ford Algorithm

If we replace the heap in Dijkstra's algorithm with a queue, we get an algorithm originally proposed by Moore in 1957 and, independently, by Bellman in 1958. The algorithm is actually called the Bellman-Ford algorithm, since the idea of relaxing edges was first proposed by Ford in 1956. The algorithm is efficient even if there are negative edges. Though Dijkstra's algorithm is faster when there are no negative edges.

```java
for (k = 0; k < V; k++) dist[k] = INFINITY;
Queue q = new Queue();
dist[s] = 0;
q.enqueue(s);
```
while (!q.isEmpty())
{
    v = q.dequeue();
    for each w in adj(v) do
    {
        if (dist[w] < dist[v] + weight[v,w])
        {
            dist[w] = dist[v] + weight[v,w];
            if (!q.isInQueue(w)) q.enqueue(w);
        }
    }
}

This algorithm is really very simple - all nodes on the queue are dequeued, processed and possibly enqueued again and again. Nevertheless, there are no substantially better algorithms.

It is essential to test that an element is not on the queue before enqueuing it, otherwise the capacity of the queue may be exceeded. The algorithm terminates if and only if a graph has no a negative cycle.

**Question.** What is running time of this algorithm?

**Exercise.** Modify the above algorithm so it terminates if a graph has a negative cycle. *Hint*, count how many times a node was enqueued. If it is more than $V - 1$, then obviously there is a negative cycle.

### 8.4.3 All-pairs shortest paths

We want to find the length of the shortest path between all pairs of vertices. One solution is to run Dijkstra $V$ times, once with each vertex as the source.

$$O(V^2 \log V + V E) = O(V^3)$$

However, if a graph has a negative edge we have to replace Dijkstra by the slower Bellman-Ford algorithm with the running time

$$O(V^2 E) = O(V^4)$$

The solution is provided by the Floyd-Warshall algorithm - dynamic programming algorithm that runs in $O(V^3)$

**The Floyd-Warshall algorithm**

To get a dynamic programming algorithm, we need first to come up with a recursive formulation.

$$\text{dist}(x, y) = \min_u (\text{dist}(x, u) + c(u, y))$$

To find the shortest path from $x$ to $y$, try all possible predecessors $u$ and compute the shortest path from $x$ to $u$ and then add the edge $(u, y)$. Unfortunately, the above idea does not work. Computing $\text{dist}(x, y)$ requires computing $\text{dist}(x, u)$ that in turn requires computing $\text{dist}(x, a)$ and so on...We need to break this infinite loop of calls.

The Floyd-Warshall algorithm starts by numbering the vertices of the graph from 1 to $n$. We use these numbers here not to label the vertices, but to order them. Define
$D(i, j)^k$ to be the length of the shortest simple path from $i$ to $j$ using only vertices numbered from 1, 2, ..., $k$ as possible intermediate vertices.

When $k = 0$, we are allowed no intermediate vertices, so that every path consists of at most one edge. In general, adding a new vertex as a possible intermediary helps only if there is a short path that goes through it, so

$$D(i, j)^k = \min(D(i, j)^{k-1}, D(i, k)^{k-1} + D(k, j)^{k-1})$$

**Implementation.**

Three nested for-loops in $i, j$ and $k$ that run from 1 to $V$.

**Question.** How does the Floyd-Warshall algorithm handle a negative cycle?

**Answer.** At the end of each iteration of the outer loop, we check the value of $D(i, i)^k$ for all $i$. If any of these are negative, then a negative cycle has been found.

**Exercise.** Compare the Floyd-Warshall and the Bellman-Ford algorithms. You can apply both of them to graphs with negative weights. Which one has a better running time?

**Answer.** This depends on a graph. For sparse graphs Bellman-Ford may be better.

**Exercise.** Design an algorithm that find a simple shortest cycle in a graph. A simple cycle is one that visits no edge or vertex twice.

**Answer.** To find the shortest simple cycle, the easiest approach is to compute the lengths of the shortest paths from $i$ to all other vertices, and then explicitly check whether there is an acceptable edge from each vertex back to $i$. 