CS 3345 Graphs

Graphs are fundamental structures in Computer Science. We will study many types:

Undirected	The edges are all bidirectional
Directed	All edges are directed
Unweighted	All edges have weight 1
Weighted	all edges have weights

Vertices contain keys or ID numbers and edges represent connections or relationships. Street maps, prerequisite charts, and electrical circuits are simple examples of graphs.

Formally, $G = (V, E)$ where $V = \{v_i$ such that v_i is a vertex $\}$,
$E = \{(i, j)$ such that $i \in V$ and $i \in V\}$.

$|V|$ is the number of vertices, often referred to by the symbol n.
$|E|$ is the number of edges, often referred to by the symbol e.

$|E| < |V|^2$.

A graph may be **completely connected**, in which every vertex is connected to every other vertex and $|E| = |V|(|V| - 1)/2$.

A dense graph has $O(|V|^2)$ edges. A sparse graph has $O(|V|)$ edges.
A graph may be cyclic or non-cyclic. A directed or undirected path that visits any node more than once constitutes a cycle.

There are many other terms used in graph theory and we shall come across some of them in the following.

Data Structures for Graphs

Adjacency List Structure

The directed graph below may be represented by an array of linked lists, one linked list for each vertex, containing the neighbors of that vertex:

A	D → E
B	E → F
C	G
D	H → J
E	J
F	J → K
G	K
H	L
J	L → M
K	M
L	
M	

The adjacency list structure is most suited to sparse graphs.

Adjacency Matrix Structure

The adjacency matrix is a 2 dimensional array or size $|V|$ by $|V|$. The elements of the array contain the edge weights. Here is the matrix for the directed graph below:

	A	B	C	D	E	F	G	H	J	K	L	M
A	0	0	0	1	1	0	0	0	0	0	0	0
B	0	0	0	0	1	1	0	0	0	0	0	0
C	0	0	0	0	0	0	1	0	0	0	0	0
D	0	0	0	0	0	0	0	1	1	0	0	0
E	0	0	0	0	0	0	0	0	1	0	0	0
F	0	0	0	0	0	0	0	0	1	1	0	0
G	0	0	0	0	0	0	0	0	0	1	0	0
H	0	0	0	0	0	0	0	0	0	0	1	0
J	0	0	0	0	0	0	1	0	0	0	1	1
K	0	0	0	0	0	0	0	0	0	0	0	1
L	0	0	0	0	0	0	0	0	0	0	0	0
M	0	0	0	0	0	0	0	0	0	0	0	0

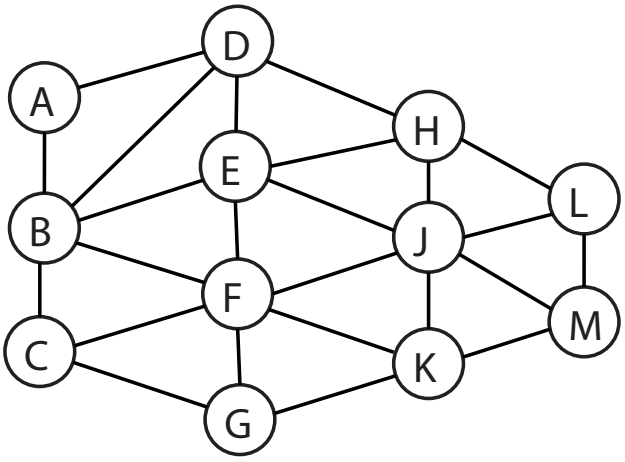
The adjacency matrix is best suited to dense graphs, but some algorithms require this structure for any graph. This structure implies at least an $O(|V|^2)$ time complexity for any algorithm that employs it. The cells of the array would contain weights in a weighted graph.

Shortest Paths for an unweighted graph

We often need to compute minimal paths from a distinguished vertex to all others. This is called the **Single Source Shortest Path Problem**.

The graph could be directed or not. The best algorithm for an unweighted graph is **Breadth First Search**.

Consider the shortest paths from source vertex s = A.



Recall the BFS algorithm, with start vertex s:

```
.....
    Queue que = new Queue()
    for each vertex u ∈ G.V − {s}
        u.color = WHITE
        u.d = ∞
        u.π = NIL
    s.color = GRAY
    s.d = 0
    s.π = NIL
    que.enque(s)
    while(que.notEmpty())
        u = que.dequeue()
        for each v ∈ G.Adj[u]
            if v.color == WHITE
                v.color = GRAY
                v.d = v.d + 1
                v.π = u
                que.enqueue(v)
        u.color = BLACK
```

Each vertex v has class elements $v.color \in \{WHITE, GRAY, BLACK\}$, $v.d$ that records the shortest distance from s, and $v.\pi$ which records the predecessor vertex on the path from s.

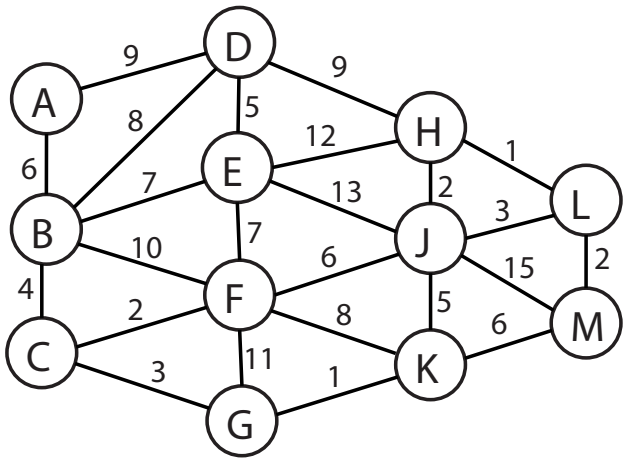
BFS for the above undirected, unweighted graph above with start vertex A reveals the following:
Distance 1 from A: B, D
Distance 2 from A: C, E, F, H
Distance 3 from A: G, J, K, L
Distance 4 from A: M

The running time of BFS is $O(|E| + |V|)$ if an adjacency list structure is used. The algorithm visits every vertex once and considers every edge twice (once from each end).

Single Source Shortest Paths in a Weighted Graph
Dijkstra's Algorithm

For weighted, directed or undirected graphs, Dijkstra's algorithm is used (unless there are negative edge weights). Again, the algorithm operates from a distinguished start vertex and computes shortest paths to all other vertices. For a sparse graph the adjacency matrix is used.

See the following weighted undirected graph:



The algorithm maintains a set of "known" vertices, S, for which the shortest path has already been computed. One vertex is added to S on each iteration.

Once a vertex is added to the S, the cost of the path to that vertex, and the shortest path, will not change in any subsequent iteration. The algorithm is "greedy" in this sense. At each iteration, the algorithm adds the vertex to S that is a a neighbor of a vertex in S, and has the shortest distance from the start vertex among all vertices not currently in S.

Here is the algorithm:

.....

Dijkstra(G)
$S = \phi$
PriorityQueue pque = new PriorityQueue(V)
While (pque.notEmpty())
    u = pque.deleteMin()
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX(u, v)

S is the set of known vertices. It is initially empty.

The vertices are inserted into the priority queue in order of their distances from the start vertex {s}. For non-neighbors of s, ∞ is inserted.

For the above weighted graph with start vertex s = A, once A has been added to S, the distances would be A.d = 0, B.d = 6, D.d = 9. $v.d = \infty$ for all the other vertices.

On each iteration the vertex u is added to the known set. It is the nearest unknown vertex to s at that time, based on the $vertex.d$ values.

The function RELAX(u, v) is as follows:

```
.....................
    RELAX(u, v)
    if v.d > u.d + w(u, v)
        v.d = u.d + w(u, v)
        v.π = u
```

The function $w(u, v)$ returns the weight of the edge (u, v). Do not assume that w implies an adjacency matrix structure. It is usually implemented by iterating along the adjacency list for vertex u.

Recall that when RELAX is called the algorithm has just added u to the known set S. The **for** loop extracts all the unknown neighbors of u. At this time, the distances recorded for these vertices are via paths that only include known vertices, but excluding u (since we have only just added u).

For each v in this subset the algorithm checks to see if the current path to v can be improved by going via u.

Dry-running the algorithm for the above weighted graph yield the following first four iterations:

v	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$
A	F	∞	ϕ	T	0	-	T	0	-	T	0	-	T	0	-
B	F	∞	ϕ	F	6	A	T	6	A	T	6	A	T	6	A
C	F	∞	ϕ	F	∞	ϕ	F	10	B	F	10	B	T	10	B
D	F	∞	ϕ	F	9	A	F	9	A	T	9	A	T	9	A
E	F	∞	ϕ	F	∞	ϕ	F	13	B	F	13	B	F	13	B
F	F	∞	ϕ	F	∞	ϕ	F	16	B	F	16	B	F	12	C
G	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	13	C
H	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	18	D	F	18	D
J	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
K	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
L	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
M	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
	S empty			A added to S			B added to S			D added to S			C added to S		

The next vertex to be added to S will be F and its distance F.d will be frozen at its current value of 12. Next, the unknown neighbors of F will be considered for relaxation. These are E, G, J, and K. J will be relaxed. J.d = ∞ and F.d + w(F,J) = 12 + 6 = 18. J.π will be updated to F. Similarly K will be relaxed from K.d = ∞ to F.d + w(F,K) = 12 + 8 = 20. K.π will be updated to F. All other distances will remain the same.

The final result is below:

v	S	$v.d = \delta(s, v)$	$v.\pi$	Shortest Path
A	T	0	A	A
B	T	6	A	A B
C	T	10	B	A B C
D	T	9	A	A D
E	T	13	B	A B E
F	T	12	C	A B C F
G	T	13	C	A B C G
H	T	18	D	A D H
J	T	18	F	A B C F J
K	T	14	G	A B C G K
L	T	19	H	A D H L
M	T	20	K	A B C G K M

The $v.d$ values give the shortest path costs, $\delta(s, v)$

The shortest paths are discovered by backtracking using the $v.\pi$ values.

Running Time of Dijkstra's Algorithm

Initializing the priority queue takes $O(|V|)$ time by a call to buildHeap().

There are $|V|$ iterations. In each iteration we:

1. call $x =$ deleteMin() on the priority queue
2. relax all the neighbors of x

If a binary heap is used as the priority queue, deleteMin() takes $O(log|V|)$ time.

If an adjacency list is used to store the edges of G, then the algorithm can step through the list of neighbors of x during relaxation. For each neighbor y of x, a relaxation that changes $y.d$ will require a call to decreaseKey(y). Each decreaseKey() takes $O(log|V|)$. For each new vertex x added to S, the relaxation takes $O(x.degree \times log|V|)$ where $x.degree$ is the number of neighbors of x. Overall, the algorithm takes:

$$\sum_{v \in G} (1 + v.degree) log|V|$$

which is

$$O(|V| + |E|) log|V|$$

For a dense graph, $|E| = O(|V|^2$ and the running time is $O(|V|^2 log|V|)$.

For sparse graphs, $|E| = O(|V|)$ and the running time is $O(|V| log|V|)$.

A simpler implementation of the algorithm makes use of an array instead of a priority queue to store the $v.d$ values.

The equivalent of deleteMin() in an array requires a linear search to find (but not remove) the minimum value, taking $O(|V|)$ time. If an adjacency list is used, finding each neighbor of a vertex

takes constant time. Relaxing vertex v also takes constant time (we simply adjust the value of $v.d$ in the distance array).

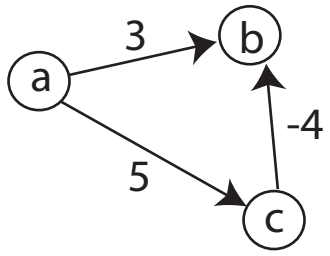
Overall the algorithm takes

$$\sum_{v \in G} (|V| + degree(v)) = O(|V|^2 + |E|)$$

For a sparse graph or a dense graph the running time is $O(|V|^2)$. For a dense graph, this algorithm is optimal since it runs in a time linear in the number of edges (it beats the implementation above using a binary heap).

A Fibonacci Heap provides amortized constant time for decreaseKey(), giving $O(|E| + |V|log|V|)$ runtime for Dijkstra's algorithm for sparse graphs.

Negative edge weights
Consider the following directed, weighted graph:



Beginning with the start vertex a, Dijkstra's algorithm first adds a and then b to S. After b is added to S, its shortest path distance from a is fixed at 3. But after c is added, the negative edge (c, b) would provide an opportunity to update $a's$ distance to 1. This shows that Dijkstra's greedy algorithm should never be used with a graph containing negative edges.

Floyd Warshall All Pairs Shortest Path Algorithm

This algorithm computes in $\theta(|V|^3)$ time the shortest path between every pair of vertices. The algorithm is simple, comprising three nested loops. Its input is a $|V| \times |V|$ adjacency matrix structure, W. The algorithm computes a $|V| \times |V|$ array d of shortest path distances and a $|V| \times |V|$ array $path$ containing the predecessor vertices on the shortest paths.

```
.....
    FloydWarshall(W)
    n = W.rows
//Initialize d and path
    for i = 0 to n − 1
        for j = 0 to n − 1
            d_{i,j} = W_{i,j} // W_{i,j} is assumed to be 0
            path_{i,j} = φ
    for k = 0 to n − 1
```

```
for i = 0 to n − 1
    for j = 0 to n − 1
        if d_{i,k} + d_{k,j} < d_{i,j}
            d_{i,j} = d_{i,k} + d_{k,j}
            path_{i,j} = k
```

Although the algorithm takes $O(|V|^3)$ time, the loops are tight and, for small dense graphs, Floyd Warshall is often used in place of Dijkstra's single source shortest path algorithm.

The algorithm proceeds by considering each edge (i, j) and checking if its shortest path might be improved by going via an intermediate vertex k: $d_{i,k} + d_{k,j} < d_{i,j}$. This process is repeated for all $k = 0, 1, 2, \cdots, |V| - 1$.

The largest value in the array d gives the **Diameter** of the graph. This is the largest of the shortest distances.

Minimal Spanning Tree

A minimal spanning tree is defined for an undirected, weighted graph $G(V, E)$. It is a set of $|V| - 1$ edges from E that induces a tree on the vertices in V. The total weight of all the tree edges is minimal. When the edge weights of G are not unique there may be several equally good minimal spanning trees.

Prim's Algorithm

Prim 's algorithm is very similar to Dijkstra's single source shortest path algorithm. The relaxation method is different. Here is the algorithm with start vertex s:

```
.....
    Prim(G,s)
    Set S = new Set()
    S = φ
    PriorityQueue pque = new PriorityQueue(V)
    for each u ∈ G.V
        u.d = ∞
        u.π = NIL
    r.d = 0
    while pque.notEmpty()
        u = pque.deleteMin()
        S = S ∪ u
        for each v ∈ G.Adj[u]
            if (v ∉ S) and w(u, v) < v.d
                v.π = u
                v.d = w(u, v)
```

In the relaxation sequence (the last 4 lines) only vertices that are neighbors of u and not members of S are relaxed. Note that all vertices are either in S or in the priority queue. If the priority queue

has a "contains" function then the set S can be eliminated.

Once a vertex u is selected by a call to deleteMin(), its distance $u.d$ will never change again. In that sense the algorithm is greedy. Selected vertices are members of the growing spanning tree.

Careful inspection will show that when vertex u is added to S, the distance $u.d$ is the shortest distance between any vertex not in S and any member of S at that time.

Here is a dry-run of Prim's algorithm on the undirected graph above with $s = A$ as the start vertex.

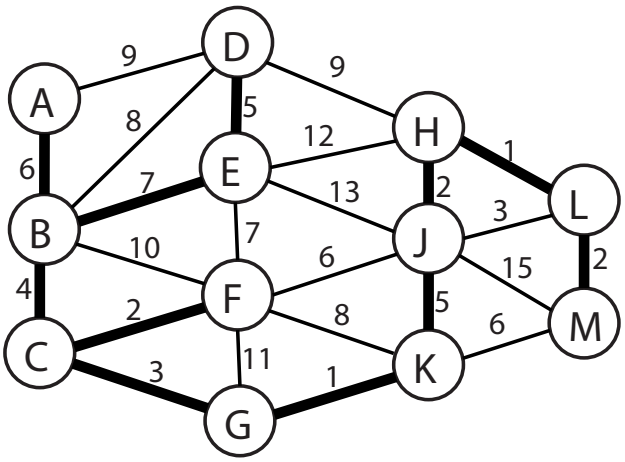
v	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$	S	$v.d$	$v.\pi$
A	F	∞	ϕ	T	0	-	T	0	-	T	0	-	T	0	-
B	F	∞	ϕ	F	6	A	T	6	A	T	6	A	T	6	A
C	F	∞	ϕ	F	∞	ϕ	F	4	B	T	4	B	T	4	B
D	F	∞	ϕ	F	9	A	F	8	A	F	8	B	F	8	B
E	F	∞	ϕ	F	∞	ϕ	F	7	B	F	7	B	F	7	B
F	F	∞	ϕ	F	∞	ϕ	F	10	B	F	2	C	T	2	C
G	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	3	C	F	3	C
H	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
J	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	6	F
K	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	8	F
L	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
M	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ	F	∞	ϕ
	S empty			A added to S			B added to S			C added to S			F added to S		

G will be selected next and its neighbors that aren't already in S will be relaxed. G only has one neighbor not in S at this time, and that is K. K.d is updated to 1 and K.π becomes G.

The final result is below:

v	S	$v.d$	$v.\pi$	edges
A	T	0	A	(A,B)
B	T	6	A	(B,C)
C	T	4	B	(B,E)
D	T	5	E	(C,F)
E	T	7	B	(C,G)
F	T	2	C	(D,E)
G	T	3	C	(G,K)
H	T	2	J	(H,J)
J	T	5	K	(H,L)
K	T	1	G	(J,K)
L	T	1	H	(L,M)
M	T	2	L	

Here is the resulting minimal spanning tree:



The running time of Prim's algorithm is identical to that of Dijkstra. An adjacency list structure should be used. Then the running time is $O(|V|^2)$ if an array is used to store the $v.d$ distance values. This is optimal for dense graphs.

If a binary heap is used to store the $v.d$ values, then the running time is $O(|E|log|V|)$, which is a better result for sparse graphs than when an array is used.

If a Fibonacci Heap is used for storing $v.d$ values, the running time of Prim's algorithm is $O(|E| + |V|log|V|)$, which is optimal for sparse graphs.