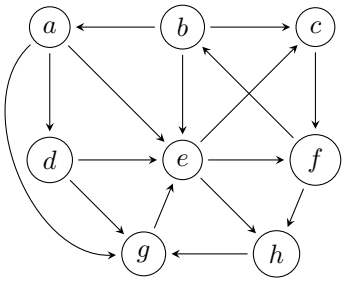
1 Theory

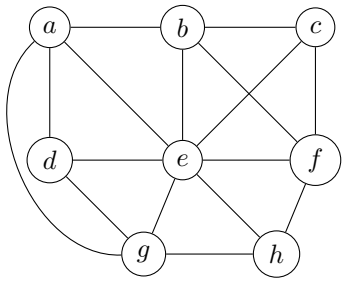
1. (*5 Points*) Draw the representation of this following graph as an adjacency list, as if it were a linked list of linked lists.



Solution:

a	$\rightarrow$	d	$\rightarrow$	e	$\rightarrow$	g
b	$\rightarrow$	a	$\rightarrow$	c	$\rightarrow$	e
c	$\rightarrow$	f				
d	$\rightarrow$	e	$\rightarrow$	g		
e	$\rightarrow$	c	$\rightarrow$	f	$\rightarrow$	h
f	$\rightarrow$	b	$\rightarrow$	h		
g	$\rightarrow$	e				
h	$\rightarrow$	g				

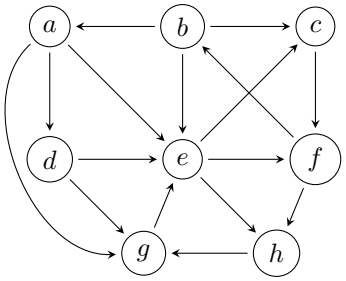
2. (*5 Points*) Draw the representation of this following graph as an adjacency matrix. Assume the row and column designations for the vertices are in alphabetical order.



Solution:

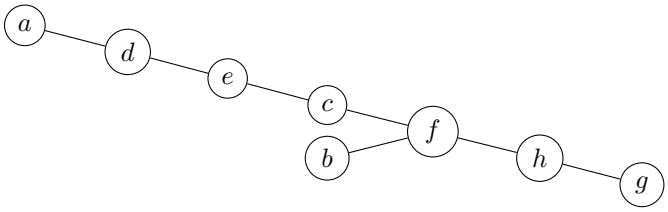
$$\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

3. (*5 Points*) Create the spanning tree of the following graph using a Depth First Search. As usual, process the nodes and edges in alphabetical order.

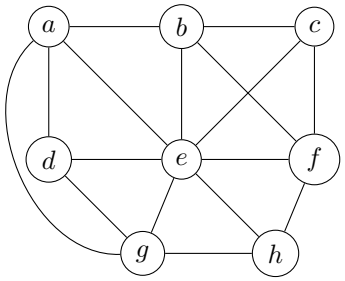


Solution:

Edges: $a-d$, $d-e$, $e-c$, $c-f$, $f-b$, $f-h$, $h-g$.

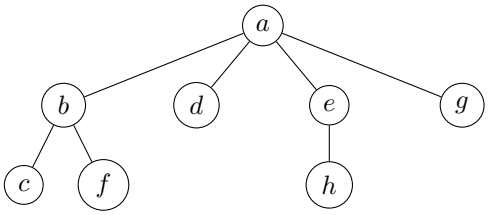


4. (*5 Points*) Create the spanning tree of the following graph using a Breadth First Search. As usual, process the nodes and edges in alphabetical order.

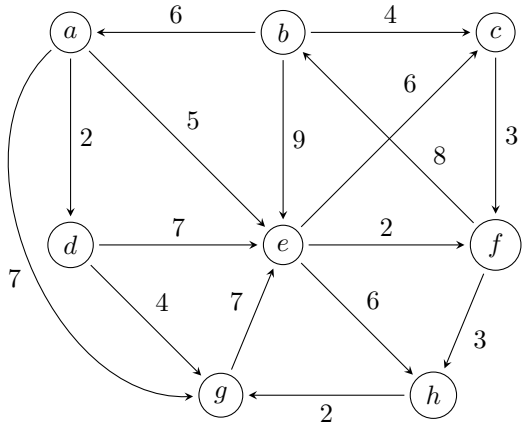


Solution:

Edges: $a-b$, $a-d$, $a-e$, $a-g$, $b-c$, $b-f$, $e-h$.



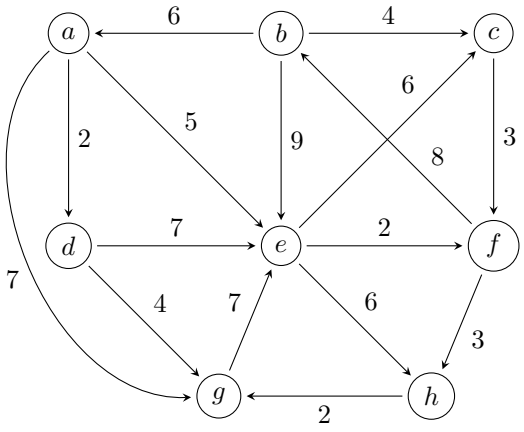
5. (*10 Points*) Go through all the steps of Dijkstra's Algorithm to determine the shortest path from vertex a to all other vertices. Create a chart of the steps as we did in class and is represented in the text.



Solution:

Active Vertex		a	d	e	g	f	h	c	b
a	0								
b	∞	∞	∞	∞	∞	15	15	15	
c	∞	∞	∞	11	11	11	11		
d	∞	2							
e	∞	5	5						
f	∞	∞	∞	7	7				
g	∞	7	6	6					
h	∞	∞	∞	11	11	10			

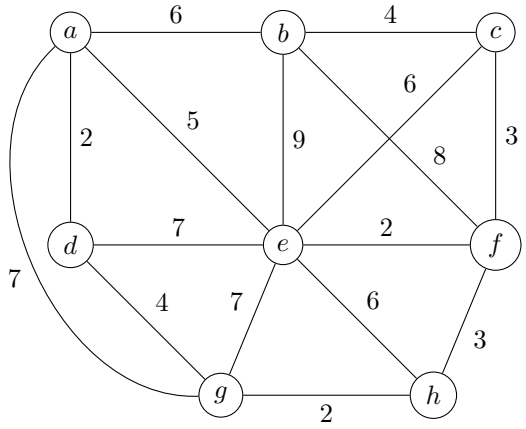
6. (*10 Points*) Go through all the steps of Ford's Algorithm to determine the shortest path from vertex a to all other vertices. Create a chart of the steps as we did in class and is represented in the text. As usual, process the nodes and edges in alphabetical order.



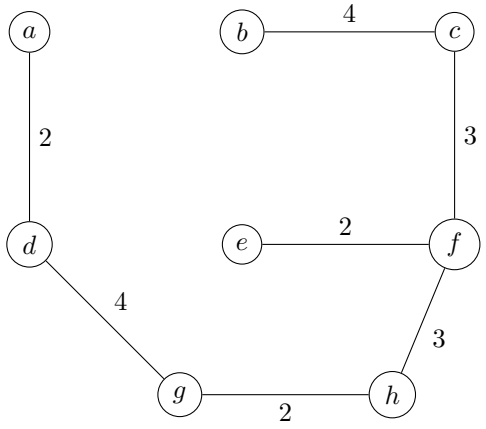
Solution:

Iteration		1		2
a	0	0		0
b	∞	15		15
c	∞	11		11
d	∞	2		2
e	∞	5		5
f	∞	7		7
g	∞	7	6	6
h	∞	11	10	10

7. (*10 Points*) Go through all the steps of Kruskal's Algorithm to determine the minimal spanning tree of the following graph.



Solution:



2 Coding

1. (*25 Points*) Code any two sorts from the following list.

   (a) Merge Sort
   (b) Quick Sort
   (c) Comb Sort
   (d) Shell Sort
   (e) Radix Sort for non-negative integer data.
   (f) Count Sort for positive integer data.
   (g) Bucket Sort for floating point data in the range $[0, 1)$.

Solution:

```
template<class T>
void merge(T A[], T Temp[], int startA, int startB, int end) {
    int aptr = startA;
    int bptr = startB;
    int i = startA;

    while (aptr < startB && bptr <= end)
        if (A[aptr] < A[bptr])
            Temp[i++] = A[aptr++];
        else
            Temp[i++] = A[bptr++];

    while (aptr < startB)
        Temp[i++] = A[aptr++];

    while (bptr <= end)
        Temp[i++] = A[bptr++];

    for (i = startA; i <= end; i++)
        A[i] = Temp[i];
}

template<class T>
void mergeSort(T A[], T Temp[], int start, int end) {
    if (start < end) {
        int mid = (start + end) / 2;
        mergeSort(A, Temp, start, mid);
        mergeSort(A, Temp, mid + 1, end);
        merge(A, Temp, start, mid + 1, end);
    }
}

template<class T>
void mergeSort(T A[], int size) {
    T *Temp = new T[size];
    mergeSort(A, Temp, 0, size - 1);
    delete[] Temp;
}

//////////////////////////////////////////////////////////

template<class T>
void quickSort(T A[], int left, int right) {
    int i = left;
    int j = right;
    int mid = (left + right) / 2;

    T pivot = A[mid];
```

```
    while (i <= j) {
        while (A[i] < pivot)
            i++;

        while (A[j] > pivot)
            j--;

        if (i <= j) {
            T tmp = A[i];
            A[i] = A[j];
            A[j] = tmp;
            i++;
            j--;
        }
    }

    if (left < j)
        quickSort(A, left, j);

    if (i < right)
        quickSort(A, i, right);
}

template<class T>
void quickSort(T A[], int size) {
    quickSort(A, 0, size - 1);
}

//////////////////////////////////////////////////////////

template<class T>
void combsort(T data[], const int n) {
    int step = n;
    while ((step = int(step / 1.3)) > 1)
        for (int j = n - 1; j >= step; j--) {
            int k = j - step;
            if (data[j] < data[k])
                swap(data[j], data[k]);
        }
    bool again = true;
    for (int i = 0; i < n - 1 && again; i++)
        for (int j = n - 1, again = false; j > i; --j)
            if (data[j] < data[j - 1]) {
                swap(data[j], data[j - 1]);
                again = true;
            }
}

//////////////////////////////////////////////////////////

template<class T>
void Shellsort(T data[], int n) {
    int i, j, hCnt, h;
    int increments[20], k;
    for (h = 1, i = 0; h < n; i++) {
        increments[i] = h;
        h = 3 * h + 1;
    }
    for (i--; i >= 0; i--) {
        h = increments[i];
        for (hCnt = h; hCnt < 2 * h; hCnt++) {
            for (j = hCnt; j < n;) { // array data
                T tmp = data[j];
                k = j;
                while (k - h >= 0 && tmp < data[k - h]) {
```

```
                    data[k] = data[k - h];
                    k -= h;
                }
                data[k] = tmp;
                j += h;
            }
        }
    }
}

//////////////////////////////////////////////////////////

template<class T>
void radixsort(T data[], const int n, const int radix) {
    int d, j, k;
    T factor;
    deque<T> queues[radix];
    for (d = 0, factor = 1; d < radix; factor *= radix, d++) {
        for (j = 0; j < n; j++)
            queues[(data[j] / factor) % radix].push_back(data[j]);
        for (j = k = 0; j < radix; j++)
            while (!queues[j].empty()) {
                data[k++] = queues[j].front();
                queues[j].pop_front();
            }
    }
}

//////////////////////////////////////////////////////////

template<class T>
void countsort(T *A, long sz) {
    T maxval = A[0];
    for (int i = 0; i < sz; i++)
        maxval = max(maxval, A[i]);

    int *counts = new int[maxval + 1]();
    int *temp = new int[sz];

    for (int i = 0; i < sz; i++)
        counts[A[i]]++;

    for (int i = 1; i < maxval + 1; i++)
        counts[i] += counts[i - 1];

    for (int i = 0; i < sz; i++)
        temp[--counts[A[i]]] = A[i];

    copy(temp, temp + sz, A);
    delete[] temp;
    delete[] counts;
}

//////////////////////////////////////////////////////////

template<class T>
void BucketSort(T *A, long sz) {
    vector<T> *Buckets = new vector<T> [sz];

    for (long i = 0; i < sz; i++)
        Buckets[static_cast<long>(sz * A[i])].push_back(A[i]);

    for (long i = 0; i < sz; i++)
        insertion(Buckets[i].data(), Buckets[i].size());
```

```
    int pos = 0;
    for (long i = 0; i < sz; i++)
        for (long j = 0; j < Buckets[i].size(); j++)
            A[pos++] = Buckets[i][j];

    delete[] Buckets;
}
```

2. (*25 Points*) Given the following code framework, code any two of the following. All implementations are to be templated.

   (a) Depth first search of the given graph. This will print out a list of edges to the console in order of the edge traversal for this algorithm. The graph is stored as a ListOfLists object in adjacency list form (not an adjacency matrix).

   `void depthFirstSearch(ListOfLists<T> G)`

   (b) Breadth first search of the given graph. This will print out a list of edges to the console in order of the edge traversal for this algorithm. The graph is stored as a ListOfLists object in adjacency list form (not an adjacency matrix).

   `void breadthFirstSearch(ListOfLists<T> G)`

   (c) Dijkstra's algorithm to find the shortest path. This will return a vector of weighted nodes that will store the node name and the minimal distance from the start node to the node itself. The graph is stored as a list (vector) of weighted edge objects. The parameter start is the initial vertex to start from.

   `vector<wnode<T>> DijkstraAlgorithm(vector<wedge<T>> G, T start)`

   (d) Ford's algorithm to find the shortest path. This will return a vector of weighted nodes that will store the node name and the minimal distance from the start node to the node itself. The graph is stored as a list (vector) of weighted edge objects. The parameter start is the initial vertex to start from.

   `vector<wnode<T>> FordAlgorithm(vector<wedge<T>> G, T start)`

   (e) Kruskal's algorithm to find the minimal spanning tree. This will return a vector of weighted edges that will store the edge list for the construction of the minimal spanning tree. The graph is stored as a list (vector) of weighted edge objects.

   `vector<wedge<T>> KruskalAlgorithm(const vector<wedge<T>> &G)`

```
template<class T>
class edge {
public:
    T f, t;

    edge(T from, T to) {
        f = from;
        t = to;
    }
};

template<class T>
class wnode {
public:
    T name;
    double weight;

    wnode(T t, double w = 0) {
        name = t;
        weight = w;
    }
```

```
    friend ostream& operator <<(ostream &strm, const wnode &obj) {
        strm << obj.name << " : " << obj.weight;
        return strm;
    }

};

template<class T>
class wedge {
public:
    T from, to;
    double weight;

    wedge(T f, T t, double w = 0) {
        from = f;
        to = t;
        weight = w;
    }

    bool operator<(const wedge &rhs) {
        return weight < rhs.weight;
    }

    bool operator>(const wedge &rhs) {
        return weight > rhs.weight;
    }

    bool operator==(const wedge &rhs) {
        return (weight == rhs.weight) && (from == rhs.from) &&
        (to == rhs.to);
    }

    friend ostream& operator <<(ostream &strm, const wedge &obj) {
        strm << obj.from << " -> " << obj.to << " : " << obj.weight;
        return strm;
    }
};
```

You may also assume that you have the `ListOfLists` structure we used in class, specification is below.

```
template<class T>
class ListOfLists {
protected:
    vector<vector<T>> list;

public:
    ListOfLists(int rows = 0, int cols = 0);
    virtual ~ListOfLists();

    int size();
    void addRow();
    void addRows(int rows = 1, int cols = 0);
    void push_back(vector<T>);
    vector<T>& operator[](const int&);
};
```

You may also assume that you have the following functions at your disposal without creating them.

- `int findVertexPos(ListOfLists<T> G, T v)` returns the position of the vertex v in the ListOfLists structure for graph G.
- `int getWnodePos(const vector<wnode<T>> &nodes, T node)` returns the position of the weighted node *node* in the list of weighted nodes *nodes*.

- `int findMinWnodePos(const vector<wnode<T>> &nodes)` returns the position of the minimum weighted node in the list of weighted nodes *nodes*.
- `bool detectCycles(ListOfLists<T> G)` that will return true if there is a cycle in the graph and false otherwise.

Solution:

```
template<class T>
void depthFirstSearch(ListOfLists<T> G) {
    vector<int> num;
    vector<edge<T>> E;
    int count = 1;

    for (int i = 0; i < G.size(); i++)
        num.push_back(0);

    while (find(num.begin(), num.end(), 0) < num.end()) {
        int pos = find(num.begin(), num.end(), 0) - num.begin();
        DFS(G, num, pos, count, E);
    }

    for (edge<T> e : E)
        cout << e.f << " - " << e.t << endl;
}

template<class T>
void DFS(ListOfLists<T> G, vector<int> &num, int pos, int &count,
        vector<edge<T>> &E) {
    vector<T> Adj = G[pos];
    num[pos] = count++;

    for (int i = 1; i < Adj.size(); i++) {
        T vert = Adj[i];

        int vPos = findVertexPos(G, vert);
        if (vPos >= 0 && num[vPos] == 0) {
            E.push_back( { G[pos][0], vert });
            DFS(G, num, vPos, count, E);
        }
    }
}

////////////////////////////////////////////////////

template<class T>
void breadthFirstSearch(ListOfLists<T> G) {
    vector<int> num;
    vector<T> V;
    vector<edge<T>> E;
    deque<T> queue;
    int count = 1;

    for (int i = 0; i < G.size(); i++) {
        num.push_back(0);
        V.push_back(G[i][0]);
    }

    while (find(num.begin(), num.end(), 0) < num.end()) {
        int pos = find(num.begin(), num.end(), 0) - num.begin();
        num[pos] = count++;
        queue.push_back(G[pos][0]);
        while (!queue.empty()) {
            T vert = queue.front();
```

```
                queue.pop_front();
                int vPos = findVertexPos(G, vert);
                vector<T> Adj = G[vPos];
                for (int i = 1; i < Adj.size(); i++) {
                    int AdjvPos = findVertexPos(G, Adj[i]);
                    if (num[AdjvPos] == 0) {
                        num[AdjvPos] = count++;
                        queue.push_back(Adj[i]);
                        E.push_back( { vert, Adj[i] });
                    }
                }
            }
        }

        for (edge<T> e : E)
            cout << e.f << " - " << e.t << endl;
    }

    ////////////////////////////////////////////////////

    template<class T>
    void DijkstraInit(const vector<wedge<T>> &G, vector<wnode<T>> &nodes) {
        double max = 1E100; // DBL_MAX;

        for (int i = 0; i < G.size(); i++) {
            wedge<T> e = G[i];
            bool newnode = true;
            for (int j = 0; j < nodes.size(); j++) {
                if (e.from == nodes[j].name)
                    newnode = false;
            }
            if (newnode)
                nodes.push_back( { e.from, max });

            newnode = true;
            for (int j = 0; j < nodes.size(); j++) {
                if (e.to == nodes[j].name)
                    newnode = false;
            }
            if (newnode)
                nodes.push_back( { e.to, max });
        }
    }

    template<class T>
    vector<wnode<T>> DijkstraAlgorithm(vector<wedge<T>> G, T start) {
        vector<wnode<T>> nodes;
        vector<wnode<T>> tobechecked;
        int upos, tbcpos;

        DijkstraInit(G, nodes);
        nodes[getWnodePos(nodes, start)].weight = 0;
        for (int i = 0; i < nodes.size(); i++)
            tobechecked.push_back(nodes[i]);

        while (!tobechecked.empty()) {
            tbcpos = findMinWnodePos(tobechecked);
            wnode<T> v = tobechecked[tbcpos];
            tobechecked.erase(tobechecked.begin() + tbcpos);
            v = nodes[getWnodePos(nodes, v.name)];

            for (int j = 0; j < G.size(); j++)
                if (v.name == G[j].from)
                    for (int i = 0; i < tobechecked.size(); i++)
                        if ((tbcpos = getWnodePos(tobechecked, G[j].to)) !=
```

```
                        -1) {
                         upos = getWnodePos(nodes, G[j].to);
                         if (nodes[upos].weight > v.weight + G[j].weight) {
                             nodes[upos].weight = v.weight + G[j].weight;
                             tobechecked[tbcpos].weight = nodes[upos].
                                weight;
                         }
                    }
        }

        return nodes;
    }

    ////////////////////////////////////////////////////

    template<class T>
    vector<wnode<T>> FordAlgorithm(vector<wedge<T>> G, T start) {
        vector<wnode<T>> nodes;
        int upos, vpos;

        DijkstraInit(G, nodes);
        nodes[getWnodePos(nodes, start)].weight = 0;

        bool finished = false;
        while (!finished) {
            finished = true;

            for (int j = 0; j < G.size(); j++) {
                upos = getWnodePos(nodes, G[j].to);
                vpos = getWnodePos(nodes, G[j].from);
                if (nodes[upos].weight > nodes[vpos].weight + G[j].weight) {
                    nodes[upos].weight = nodes[vpos].weight + G[j].weight;
                    finished = false;
                }
            }
        }

        return nodes;
    }

    ////////////////////////////////////////////////////

    template<class T>
    vector<wedge<T>> KruskalAlgorithm(const vector<wedge<T>> &G) {
        vector<wedge<T>> MST;
        vector<wedge<T>> MST_test;
        vector<wedge<T>> H = G;
        sort(H.begin(), H.end());
        int vcount = vertexList(H).size();

        for (int i = 0; i < H.size() && MST.size() < vcount - 1; i++) {
            wedge<T> e = H[i];
            MST_test = MST;
            MST_test.push_back(e);

            if (!detectCycles(MST_test)) {
                MST = MST_test;
            }
        }
        return MST;
    }
```