### Graphs

A highly useful data structure for modeling of maps, networks, relationships, and so forth.

- Defined by two sets:
  - A set of nodes, also called vertices
  - A set of edges that are connections linking pairs of vertices

- Chapter topics:
  - Basic graph terminology
  - Graph implementations
  - Common graph traversal (search) algorithms
  - Common graph algorithms for shortest path
  - Spanning tree algorithms

---

### Definitions

- A **Graph** $G = (V,E)$ consists of a set of vertices $V$ and a set of edges $E$, such that each edge in $E$ is a connection between a pair of vertices in $V$.
  - The number of vertices is written $|V|$ and the number of edges $|E|$.
    - $|E|$ may range from 0 up to $\Theta(|V|^2)$.
    - A **sparse** graph is one with relatively few edges.
    - A **dense** graph is one with relatively many edges.
    - A **complete** graph is one with all possible edges.

---

### Definitions

- An **undirected graph** is a graph whose edges are not directed.
  - *Example: an undirected graph*

---

### Definitions

- A **directed graph** or digraph is a graph whose edges are directed from one edge to another.
  - *Example: a directed graph*

- Example: a labeled, weighted directed graph
Definitions

- **Adjacent**: two vertices joined by an edge. They are also called neighbors.
- **Incident**: an edge connecting vertices u and v, written as \((u,v)\), is incident on u and v.
- **Path**: a path of length \(n-1\) is formed by the sequence of vertices \(v_1, v_2, \ldots, v_n\) if there exist edges from \(v_i\) to \(v_{i+1}\) for \(1 \leq i < n\).
  - **Simple path**: all vertices on the path are distinct.
  - **Length of the path**: the number of edges it contains.
  - **Cycle**: path of length 3 or more connecting some vertex to itself.
  - **Simple cycle**: a cycle that is a simple path except for the first/last vertex.

Graph Representations

- **Adjacency matrix**:
  - If \(|V| = n\), then the matrix is an \(n \times n\) array.
  - Rows are labeled 0 through \(n-1\) corresponding to vertices \(v_0\) to \(v_{n-1}\).
  - Row \(i\) contains entries for vertex \(v_i\).
  - The \((i,j)\) entry represents whether there is an edge between \(v_i\) and \(v_j\).
  - The \((i,j)\) entry can be a single bit (1 for present, 0 for absent) or a weight (some number for 'present with weight \(x\) or 0 for absent).
  - **Space requirements**: \(\Theta(|V|^2)\)
  - **Example**: a directed graph

Graph Representations

- **Adjacency list**:
  - Represented by an array of linked lists.
  - If \(|V| = n\), then the array has \(n\) entries.
  - List \(i\) represents the list of vertices adjacent to \(v_i\) in a directed sense.
  - As with the matrix, an entry can be 0 or 1 for unweighted graphs or it can have another numeric value to represent a weight.
  - **Space requirements**: \(\Theta(|V| + |E|)\)
  - **Example**: a directed graph

- In the linked-list node, the first field is the vertex label and second field a weight. The weight field is omitted if it is an unweighted graph.
Comparison of Representations

- Space efficiency: depends on the number of edges
  - Sparsely populated: adjacency list
  - Densely populated: adjacency matrix
- Time efficiency: often the adjacency list is better
  - Many algorithms require visiting of all neighbors...

Graph Implementations

- Graph abstract class
  ```cpp
class Graph {
  public:
    virtual int n() = 0;
    virtual int e() = 0;
    virtual int first(int) = 0;
    virtual int next(int, int) = 0;
    virtual void setEdge(int, int, int) = 0;
    virtual void delEdge(int, int) = 0;
    virtual int weight(int, int) = 0;
    virtual int getMark(int) = 0;
    virtual void setMark(int, int) = 0;
};
```

The Edge Class

- Abstract class for graph edges
  ```cpp
class Edge {
  int v1() = 0;
  int v2() = 0;
};
```

The Adjacency Matrix

- Adjacency Matrix Class Header:
  ```cpp
class Graphm : public Graph {
  private:
    int numVertex, numEdge;
    int **matrix;
    int *mark;
  public:
    Graphm(int numVert) {
      int i, j;
      numVertex = numVert;
      numEdge = 0;
      mark = new int[numVertex];
      for (i = 0; i<numVertex; i++)
        mark[i] = UNVISITED;
      matrix = (int**) new int*[numVertex];
      for (i = 0; i<numVertex; i++)
        matrix[i] = new int[numVertex];
      for (i = 0; i<numVertex; i++)
        for (int j = 0; j<numVertex; j++)
          matrix[i][j] = 0;
    }
  int first(int);
  int next(int, int);
  void setEdge(int, int, int);
  void delEdge(int, int);
  int weight(int, int);
  int getMark(int);
  void setMark(int, int);
};
```
The Adjacency Matrix

- **Function Implementations**

```c
int first(int v) {
    int i;
    for (i = 0; i < numVertex; i++)
        if (matrix[v][i] != 0) return i;
    return i;
}

int next(int v1, int v2) {
    int i;
    for (i = v2 + 1; i < numVertex; i++)
        if (matrix[v1][i] != 0) return i;
    return i;
}
```

CSC 375-Turner, Page 13

The Adjacency List

- **Function Implementations**

```c
void setEdge(int v1, int v2, int wgt) {
    Assert(wgt > 0, "Illegal weight value");
    if (matrix[v1][v2] == 0) numEdge++;
    matrix[v1][v2] = wgt;
}

void delEdge(int v1, int v2) {
    if (matrix[v1][v2] != 0) numEdge--;
    matrix[v1][v2] = 0;
}

int weight(int v1, int v2) {
    return matrix[v1][v2];
}

int getMark(int v) {
    return mark[v];
}

void setMark(int v, int val) {
    mark[v] = val;
}
```

CSC 375-Turner, Page 14

The Adjacency Matrix

- **Function Implementations**

```c
int first(int v) {
    Edge it;
    vertex[v] -> setStart();
    if (vertex[v] -> getValue(it)) return it.vertex;
    else return numVertex;
}

int next(int v1, int v2) {
    Edge it;
    vertex[v1] -> getValue(it);
    if (it.vertex == v2) vertex[v1] -> next();
    else {
        vertex[v1] -> setStart();
        while (vertex[v1] -> getValue(it)
            && (it.vertex <= v2))
            vertex[v1] -> next();
    }
    if (vertex[v1] -> getValue(it)) return it.vertex;
    else return numVertex;
}
```

CSC 375-Turner, Page 15

The Adjacency List

- **Function Implementations**

```c
int first(int v) {
    int i, j;
    numVertex = numVert; numEdge = 0;
    mark = new int[numVert];
    for (i = 0; i < numVertex; i++) mark[i] = UNVISITED;
    vertex = new List<Edge>()[numVertex];
    for (i = 0; i < numVertex; i++)
        vertex[i] = new List<Edge>()[numVertex];
    }

    int n();
    int e();
    int first(int);
    int next(int, int);
    void setEdge(int, int, int);
    void delEdge(int, int);
    int weight(int, int);
    void getMark(int);
    void setMark(int, int);
}
```

CSC 375-Turner, Page 16
The Adjacency List

- Function Implementations:

```c
void setEdge(int v1, int v2, int wgt) {
    Assert(wgt>=0, "Illegal weight value");
    Edge it(v2, wgt);
    Edge curr;
    vertex[v1] -> getValue(curr);
    if (curr.vertex != v2)
        for (vertex[v1] -> setStart();
             vertex[v1] -> getValue(curr);
             vertex[v1] -> next())
            if (curr.vertex == v2) break;
    if (curr.vertex == v2)
        vertex[v1] -> remove(curr);
    else numEdge++;
    vertex[v1] -> insert(it);
}

void delEdge(int v1, int v2) {
    Edge curr;
    vertex[v1] -> getValue(curr);
    if (curr.vertex != v2)
        for (vertex[v1] -> setStart();
             vertex[v1] -> getValue(curr);
             vertex[v1] -> next())
            if (curr.vertex == v2) break;
    if (curr.vertex == v2) {
        vertex[v1] -> remove(curr);
        numEdge--;
    }
}
```

CSC 375-Turner, Page 17

The Adjacency List

- Function Implementations:

```c
int weight(int v1, int v2) {
    Edge curr;
    vertex[v1] -> getValue(curr);
    if (curr.vertex != v2)
        for (vertex[v1] -> setStart();
             vertex[v1] -> getValue(curr);
             vertex[v1] -> next())
            if (curr.vertex == v2) break;
    if (curr.vertex == v2)
        return curr.weight;
    else
        return 0;
}

int getMark(int v) {
    return mark[v];
}

void setMark(int v, int val) {
    mark[v] = val;
}
```

CSC 375-Turner, Page 18

Graph Traversals

It is often useful to visit the vertices in some specific order.

- Generic Traversal Function

```c
void graphTraverse(const Graph* G) {
    for (v = 0; v < G -> n(); v++)
        G -> setMark(v, UNVISITED);
    for (v = 0; v < G -> n(); v++)
        if (G -> getMark(v) == UNVISITED)
            doTraverse(G, v);
```

- The doTraverse(G,v) function could be one of
  - Depth-first search
    - For a given vertex, recursively visit all neighbors.
  - Breadth-first search
    - For a given vertex, examine all neighbors before visiting vertices further away.
  - Topological sort
    - Laying out vertices of a DAG in a linear order (according to prerequisite relationships).

CSC 375-Turner, Page 19

Graph Traversals

- Depth-First Search

```c
void DFS(Graph* G, int v) {
    PreVisit(G, v);
    G -> setMark(v, VISITED);
    for (int w = G -> first(v);
         w < G -> n();
         w = G -> next(w))
        if (G -> getMark(w) == UNVISITED)
            DFS(G, w);
    PostVisit(G, v);
}
```

CSC 375-Turner, Page 20
Graph Traversals

- Breadth-First Search

```c
void BFS(Graph* G, int start, Queue<int>* Q) {
    int v, w;
    Q->enqueue(start);
    G->setMark(start, VISITED);
    while (Q->length() != 0) {
        Q->dequeue(v);
        PreVisit(G, v);
        for (w = G->first(v); w < G->n(); w = G->next(v, w))
            if (G->getMark(w) == UNVISITED) {
                G->setMark(w, VISITED);
                Q->enqueue(w);
            }
        PostVisit(G, v);
    }
}
```

- Recursive Topological Sort

```c
// Public function
void topsort(Graph* G) {
    int i;
    for (i = 0; i < G->n(); i++)
        if (G->getMark(i) == UNVISITED)
            tophelp(G, i);
}

// Private function
void tophelp(Graph* G, int v) {
    G->setMark(v, VISITED);
    for (int w = G->first(v); w < G->n(); w = G->next(v, w))
        if (G->getMark(w) == UNVISITED)
            tophelp(G, w);
    printout(v);
}
```

Shortest-Paths Problems

Sometimes it is useful to use a graph to find the shortest path from point A to B.

- Edges are labeled with real numbers representing weights, costs, distances, delay, etc.
- Goal is to find the smallest weighted path.
- Single-source shortest-paths problem:
  - Given a vertex s in graph G, find a shortest path from s to every other vertex in G.
  - Approach 1:
    - Add vertices to a list S in order of distance from the source.
    - Given a vertex v_i not yet in S:
      - d(s, v_i) = min_{u < S} [d(s, u) + w(u, v_i)]
      - Means: find the minimum combination of "short path from s to a vertex already in S plus a weight coming from a vertex in S to the new vertex v_i."
Single-Source Shortest-Paths Problem

- Dijkstra’s algorithm:
  ```
  void Dijkstra(Graph* G, int* D, int s) {
      int i, v, w;
      for (i = 0; i < G->n(); i++) {
          v = minVertex(G, D);
          if (D[v] == INFINITY) return;
          G->setMark(v, VISITED);
          for (w = G->first(v);
               w < G->n();
               w = G->next(v, w))
              if (D[w] > D[v] + G->weight(v, w))
                  D[w] = D[v] + G->weight(v, w);
      }
  }
  ```

- All-Pairs Shortest-Paths Problem

  - Floyd’s Algorithm:
    ```
    void Floyd(Graph* G) {
        int D[G->n()][G->n()];
        for (int i = 0; i < G->n(); i++)
            for (int j = 0; j < G->n(); j++)
                D[i][j] = G->weight(i, j);
        for (int k = 0; k < G->n(); k++)
            for (int i = 0; i < G->n(); i++)
                for (int j = 0; j < G->n(); j++)
                    if (D[i][j] > D[i][k] + D[k][j])
                        D[i][j] = D[i][k] + D[k][j];
    }
    ```

Shortest-Paths Problems

- All-Pairs shortest-paths problem:
  - Find the shortest distance between all pairs of vertices in the graph.
  - That is, for every u, v \( \in V \), calculate \( d(u, v) \)

- Try 1: run Dijkstra’s algorithm \(|V|\) times
  - Works well if the graph is sparse, but not if it is dense.

- Try 2:
  - Uses concept of k-path: any intermediate vertex on a path between vertices u and v must be labeled less than k.
  - Direct edge between u and v is a 0-path
  - \( D_k(u, v) \) is the length of the shortest k-path from v to u.
  - If that shortest k-path is already known, then
    - The \((k+1)\)-path goes through vertex k: the best path is the best k-path from v to k followed by the best k-path from k to u.
    - The \((k+1)\)-path does not go through vertex k: keep the best k-path seen before.

Minimum-Cost Spanning Trees

- A minimum-cost spanning tree (MST) of G contains the vertices of G and a subset of its edges.

- Properties:
  1. has minimum total cost measured by summing values for all of the edges in the subset.
  2. keeps the vertices connected.

- Applications:
  - find the shortest set of wires connecting circuit components
  - Connecting a set of phones to use the least amount of wire
Prim's Algorithm

- Start with any vertex $u$
  - Pick the least-cost edge connected to $u$ that doesn't create a cycle; assume that edge is $(u, v)$.
  - Add vertex $v$ and edge $(u, v)$ to the graph
  - Repeat this until all vertices of the graph have been added.
- Finding a minimum-cost vertex:
  ```c
  int minVertex(Graph* G, int* D) {
    int i, v;  // Initialize v to any unvisited vertex;
    for (i = 0; i < G -> n(); i++)
      if (G -> getMark(i) == UNVISITED) {
        v = i;
        break;
      }
    for (i = 0; i < G -> n(); i++)
      if ((G -> getMark(i) == UNVISITED) && (D[i] < D[v]))
        v = i;
    return v;
  }
  ```

CSC 375-Turner, Page 29

Kruskal's Algorithm

- Partition the set of vertices into $|V|$ equivalence classes
- Process edges in order of weight
  - An edge is added to MST (and two equivalence classes combined) if it connects two vertices in different equivalence classes.
  - Repeat until only one equivalence class exists.
  - Store edges in a min heap to process in order of weight.

CSC 375-Turner, Page 30

Kruskal's Algorithm

- The algorithm:
  ```c
  void Kruskal(Graph* G) {
    G.stree A(G -> n());
    KruskElem E(G -> e());
    int i;
    int edgenum = 0;
    for (i = 0; i < G -> n(); i++)
      for (int w = G -> first(i);
         w < G -> n();
         w = G -> next(i, w))
        E[edgenum].distance = G -> weight(i, w);
        E[edgenum].from = i;
        E[edgenum].to = w;
        edgenum++;
    minheap H(E, edgenum, edgenum);
    int numMST = G -> n();
    for (i = 0; numMST > 1; i++) {
      KruskElem temp;
      H.removeMin(temp);
      int v = temp.from;
      int u = temp.to;
      if (A.differ(v, u)) {
        A.Union(v, u);
        AddEdgeToMST(temp.from, temp.to);
        numMST--;
      }
    }
  }
  ```

CSC 375-Turner, Page 31