Topics

- Problem modeling by graphs
- Basic graph terminology
- Graph representations
- Graph traversal — BFS and DFS
- Shortest paths — one-to-any (single source) and any-to-any (all pairs)
- (All diagrams will be provided during the lecture.)
Vegetarians and Cannibals Crossing River

- Two vegetarians and two cannibals want to cross a river on a boat that can take only two persons.
- At no time must the vegetarians be outnumbered.
- How should they cross the river?
Vegetarians and Cannibals: Hint

vvccB/
vc/Bvc vv/Bcc vvc/Bc cc/Bvv
vvcB/c
c/Bvvcc
ccB/vv vcB/vc
/Bvvcc
Wolf, Goat, Cabbage, Farmer Crossing River

- Wolf eats goat if left alone, goat eats cabbage if left alone.
- The farmer can only take one object with him at a time.
- How should the farmer bring the wolf, goat, and cabbage to the other side of the river?
Wolf, Goat, Cabbage, Farmer Crossing River: Hint

FWGC/

WC/FG

FWC/G
C/FWG W/FGC

FGC/W FWG/C
G/FWC

FG/WC

/FWGC

NOI only
Decanting: \( 3 + 5 \rightarrow 1 \)

- A 3-unit bottle and a 5-unit bottle are given.

- Each bottle can be filled from a tap, emptied into a sink, poured into another bottle until itself is empty or the other is filled.

- How to obtain 1-unit of liquid?
# Decanting: Hint

<table>
<thead>
<tr>
<th>35</th>
<th>03</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>02</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>34</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>04</td>
<td></td>
</tr>
</tbody>
</table>
Decanting : BFS (To be Discussed Later)

#define N 24

int visit[4][6];

int qx[N], qy[N];

int head, tail = 0;
check( int ox, int oy, int x, int y ) {
    if( visit[x][y] ) return;
    enqueue( x, y );
    visit[x][y] = 1;
    printf( "%d %d -> %d %d\n", ox, oy, x, y );
}
enqueue( int x, int y ) {
    if( (tail+1)%N == head ) {
        printf( "queue overflow\n" ); exit( 1 );
    }
    qx[tail] = x; qy[tail] = y;
    if( ++tail >= N ) tail = 0;
}

decqueue( int *x, int *y ) {
    if( head == tail ) {
        printf( "queue underflow\n" ); exit( 1 );
    }
    *x = qx[head]; *y = qy[head];
    if( ++head >= N ) head = 0;
}
main() {
    int i, j, x, y;

    for( i = 0; i < 4; i++ )
        for( j = 0; j < 6; j++ )
            visit[i][j] = 0;

    check( 0, 0, 0, 0 );
while( head != tail ) {
    dequeue( &x, &y );
    if( x < 3 ) check( x,y, 3, y ); // fill x
    if( x < 3 && y >= 3-x ) check( x,y, 3, y-3+x );
    if( y > 0 && y < 3-x ) check( x,y, x+y, 0 );
    if( x > 0 ) check( x,y, 0, y ); // empty x

    if( y < 5 ) check( x,y, x, 5 ); // fill y
    if( y < 5 && x >= 5-y ) check( x,y, x-5+y, 5 );
    if( x > 0 && x < 5-y ) check( x,y, 0, y+x );
    if( y > 0 ) check( x,y, x, 0 ); // empty y
}
}
0 0 \rightarrow 0 0
0 0 \rightarrow 3 0
0 0 \rightarrow 0 5
3 0 \rightarrow 3 5
3 0 \rightarrow 0 3
0 5 \rightarrow 3 2
0 3 \rightarrow 3 3
3 2 \rightarrow 0 2
3 3 \rightarrow 1 5
0 2 \rightarrow 2 0
1 5 \rightarrow 1 0
2 0 \rightarrow 2 5
1 0 \rightarrow 0 1
2 5 \rightarrow 3 4
0 1 \rightarrow 3 1
3 4 \rightarrow 0 4
Basic Graph Terminology: Vertices and Edges

- Mathematically a graph $G$ consists of a vertex set $V$ and an edge set $E$:

  $$G = (V, E).$$

- An edge has two end-points, each of which must be a vertex.

- A vertex may or may not be an end-point of an edge.

- Unless explicitly specified otherwise, a graph usually means undirected graph.
Incidence

- Incidence is a relation between a vertex and an edge.

- For any vertex \( v \) and any edge \( e \),
  vertex \( v \) is incident on edge \( e \),
  or
  edge \( e \) is incident on vertex \( v \),
  if (and only if)
  \( v \) is an end-point of \( e \).
Vertex Adjacency

- For any distinct vertices $u$ and $v$,
  
  $u$ and $v$ are adjacent
  
  if (and only if)
  
  they are incident on the same edge.

- Two distinct vertices are adjacent if (and only if) they are the end-points of the same edge.
Edge Adjacency

- For any distinct edges $e$ and $f$,
  
  $e$ and $f$ are adjacent
  
  if (and only if)
  
  they are incident on the same vertex.

- Two distinct edges are adjacent if (and only if) they share an end-point.
Simple Graphs

- An edge is a loop if (and only if) its two end-points are identical.
- Two edges are parallel if (and only if) they have the same end-points.
- A graph is simple if (and only if) there are no loops and parallel edges.
Vertex Degree

- For any vertex $v$, the degree of $v$ is the number of times edges incident on $v$.

- If there are $\mu$ loops and $\nu$ (non-loop) edges incident on a vertex $v$, the degree of $v$ is

  $$d(v) = 2\mu + \nu.$$
Isolated Vertices

- A vertex is isolated if (and only if) its degree is zero.

- That is, a vertex is isolated if (and only if) no edges incident on it; equivalently, it is incident on no edges; or no edges are incident on it.
The Handshake Theorem

• For any graph $G = (V, E)$,

$$\sum_{v \in V} d(v) = 2|E|$$

• Proof:

Each edge has two end-points.

$d(v) =$ the number of times $v$ labels an end-point.

$\sum_{v \in V} d(v) =$ number of end-points.
A Corollary

- The number of odd-degree vertices is even.

- Proof:
  \[ \sum_{v \in V} d(v) = \sum_{2|d(v)} d(v) + \sum_{2 \not| d(v)} d(v) = 2|E|. \]
Applications

- Is it possible that each of a group of nine people knows exactly five others in the group?

- Is it possible to have a graph of five vertices of degrees 1, 2, 3, 4, 5?
Paths

- Let $v_0$ and $v_1$, not necessarily distinct, be vertices of a graph.

- A path from $v_0$ to $v_1$ of length $n$ is an alternating sequence of $n + 1$ vertices and $n$ edges of the form:

  $$v_0 e_1 v_1 \cdots v_{n-1} e_n v_n$$

  such that $e_i$ is incident on $v_{i-1}$ and $v_i$ for $i = 1, \ldots, n$.

- For a simple graph, since $e_i$ is completely determined by $v_i$ and $v_{i+1}$, the path may be written simply as

  $$v_0 v_1 \cdots v_{n-1} v_n$$
Cycles

• A cycle is a path of nonzero length from a vertex to itself with distinct edges.

• A length 1 cycle is a loop.

• A length 2 cycle is two parallel edges.

• A simple cycle is a cycle with distinct vertices (except the first and last).
Connectedness

• A graph $G = (V, E)$ is connected if (and only if) for any distinct vertices $u \in V$ and $v \in V$, there is a path from $u$ to $v$. 
Acyclicity

• A graph is acyclic if (and only if) it has no cycles.
Trees

- A tree is a connected acyclic graph.
Graph Representation 1: Adjacency Matrices

- Number the $n$ vertices of a graph either from 0 to $n - 1$ or from 1 to $n$.
- The graph can be represented as a matrix $a[i][j]$ such that
  \[ a[i][j] = \text{number of edges incident on vertices } i, j. \]
- For a simple graph, we have
  \[ a[i][i] = 0 \]
  and
  \[ a[i][j] \leq 1. \]
Graph Representation 2: Adjacency Lists

- Number the $n$ vertices of a graph either from 0 to $n - 1$ or from 1 to $n$.
- Create an array $a[\ ]$ of lists.
- Array entry $a[i]$ lists the vertices adjacent to vertex $i$.
- Alternatively, create a 2-dimensional jagged array $a[ ][ ]$:

  \[\text{length of } a[i] = d(i).\]

  (Very easily done in Java.)
Adjacency Matrices Versus Adjacency Lists

- **Storage:** $|V|^2$ against $|V| + 2|E|$
- **Access:** direct against serial
Example

• Represent the graph

\[ G = (\{1, 2, 3, 4, 5, 6\}, \{\{1, 2\}, \{1, 3\}, \{2, 4\}, \{2, 5\}, \{2, 6\}, \{3, 5\}\}) \]

as an adjacency matrix and as adjacency lists.

• Represent the graph

\[ G = (\{1, 2, 3, 4, 5, 6\}, \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}) \]

where \( f(e_1) = \{6\}, \ f(e_2) = \{1, 5\}, \ f(e_3) = \{1, 5\}, \ f(e_4) = \{2, 4\}, \ f(e_5) = \{3, 4\}, \ f(e_6) = \{3, 4\}, \ f(e_7) = \{1, 1\} \), as an adjacency matrix and as adjacency lists.
Breadth First Search (BFS)

- One way of traversing a graph is to do a breadth first search (BFS).

- BFS can be described as follows:

  visit a vertex
  while there is a least recently visited vertex v do
    visit all unvisited vertices adjacent to v
BFS Pseudo Code

bfs( start ) {
    for( i = 0; i < n; i++ ) visit[i] = false;
    visit[start] = true; show( start ); enqueue( start );
    while( ! empty() ) {
        for( l = adj[dequeue()]; l; l = l->next ) {
            v = l->vertex;
            if( !visit[v] ) {
                visit[v] = true; show( v ); enqueue( v );
            }
        }
    }
}
BFS: The Decanting Example

- A graph modeling the previous decanting example is given as adjacency lists.

- What is the rooted ordered tree obtained by a breadth first search starting with 00 (both bottles are empty)?

- Note that the order of the ordered tree is determined by the vertex order in the adjacency lists.
<table>
<thead>
<tr>
<th>Time</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>00:00</td>
<td>30, 05</td>
</tr>
<tr>
<td>01:00</td>
<td>31, 10, 05, 00</td>
</tr>
<tr>
<td>02:00</td>
<td>32, 20, 05, 00</td>
</tr>
<tr>
<td>03:00</td>
<td>33, 30, 05, 00</td>
</tr>
<tr>
<td>04:00</td>
<td>34, 31, 05, 00</td>
</tr>
<tr>
<td>05:00</td>
<td>35, 32, 00</td>
</tr>
<tr>
<td>10:00</td>
<td>30, 00, 15, 01</td>
</tr>
<tr>
<td>15:00</td>
<td>35, 33, 05, 10</td>
</tr>
<tr>
<td>20:00</td>
<td>30, 00, 25, 02</td>
</tr>
<tr>
<td>25:00</td>
<td>35, 34, 05, 20</td>
</tr>
<tr>
<td>30:00</td>
<td>00, 35, 03</td>
</tr>
<tr>
<td>31:00</td>
<td>01, 35, 04, 30</td>
</tr>
<tr>
<td>32:00</td>
<td>02, 35, 05, 30</td>
</tr>
<tr>
<td>33:00</td>
<td>03, 35, 15, 30</td>
</tr>
<tr>
<td>34:00</td>
<td>04, 35, 25, 30</td>
</tr>
<tr>
<td>35:00</td>
<td>05, 30</td>
</tr>
</tbody>
</table>
Depth First Search

- Another way of traversing a graph is to do a depth first search (DFS).

- DFS can be described as follows:

\[
dfs( v ) \{ \\
    \text{mark } v \text{ as visited} \\
    \text{for each unvisited vertex } u \text{ adjacent to } v \text{ do} \\
    \quad dfs( u ) \\
\}
\]
DFS Pseudo Code

dfs_init( start ) {
    for( i = 0; i < n; i++ ) visit[i] = false;
    dfs( start );
}

dfs( start ) {
    visit[start] = true; show( start );
    for( l = adj[start]; l; l = l->next ) {
        v = l->vertex;
        if( !visit[v] ) dfs( v );
    }
}
DFS: The Decanting Example

- Consider the adjacency lists of the graph for the decanting example.

- What is the rooted ordered tree obtained by a depth first search starting with 10?
DFS: The Decanting Example Answer

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>03</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>/ \</td>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>35</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>04</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Remark on the Decanting Problem

- The adjacency lists are not created explicitly and statically. They are created on the flight.

- The possible configurations from the configuration \((x, y)\):
  
  \[(x, y), x < 3 \rightarrow (3, y)\]
  
  \[(x, y), x < 3, y \geq 3 - x \rightarrow (3, y + x - 3)\]
  
  \[(x, y), y > 0, y < 3 - x \rightarrow (x + y, 0)\]
  
  \[(0, y), x > 0 \rightarrow (0, y)\]
\[(x, y), y < 5 \rightarrow (x, 5)\]
\[(x, y), y < 5, x \geq 5 - y \rightarrow (x + y - 5, 5)\]
\[(x, y), x > 0, x < 5 - y \rightarrow (0, x + y)\]
\[(x, y), y > 0 \rightarrow (x, 0)\]
Weighted Graphs

- A weighted graph $G$ consists of a vertex set $V$, an edge set $E$, and a weight function $w : E \rightarrow \mathbb{R}$:

  $$G = (V, E, w).$$

- Every graph can be treated as a weighted graph by taking

  $$w(e) = 1$$

  for any edge $e \in E$.  


Some Observations on Weighted Graphs

- Many situations can be modeled as weighted graphs.

- For example, the highways connecting cities may be modeled as a weighted graph with highway distance as the weight function.

- The rooted tree built by a breadth first search starting at vertex $v$ gives the shortest path length of all vertices from $v$: the shortest distance of a vertex at level $L$ is $L$.

- In other words, BFS solves the 1-to-any (single source) shortest path problem for the special case when $w(E) = \{1\}$. 
Weighted Graph Representations

- A simple weighted graph can be represented as an adjacency matrix:

\[ a[i][j] = w(i, j). \]

- A weighted graph can also be represented as adjacency lists:

\[ a[i] = \text{a list of pairs } (v, w(e)) \]

where \(v\) is a vertex adjacent to \(i\) and \(e\) is an edge incident on \(i\) and \(v\).
Floyd’s Algorithm: All-Pairs Shortest Path

floyd() {
    for( k = 0; k < n; k++ )
        for( i = 0; i < n; i++ )
            for( j = 0; j < n; j++ )
                if( a[i][k] + a[k][j] < a[i][j] )
                    a[i][j] = a[i][k] + a[k][j];
}
Floyd’s Algorithm: Comments

- Very easy to code.
- Transform the adjacency matrix to an all-pairs shortest path matrix.
- Complexity $\Theta(n^3)$.

Theory: dynamic programming subproblem structure:

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

Programming: update in place is correct because

$$a^{(k)}_{i,k} = a^{(k-1)}_{i,k}, \quad a^{(k)}_{k,j} = a^{(k-1)}_{k,j}.$$
Dijkstra’s Algorithm for Single-Source Shortest Paths

- This is a greedy algorithm.
- Theory (skipped).
Dijkstra’s Algorithm: Pseudo Code

for each vertex \( v \) do \( d[v] = \text{MAX}\_\text{VAL} \);
\( d[\text{source}] = 0; \quad p[\text{source}] = -1; \)
do \(|V|\) times {
    let \( d[v] \) be the smallest among all undeleted vertices
    \( x = d[v]; \quad \text{delete} \ v; \)
    for each vertex \( u \) adjacent to \( v \) do {
        if \( u \) is undeleted and \( x + w[u] < d[u] \) then {
            \( d[u] = x + w[u]; \quad p[u] = v; \)
        }
    }
} // array \( d \) gives the shortest distance from source
Implementation: Naive Versus Sophisticated

- Two pieces of information: adjacency and distance.

- If the distance information is implemented as an array $d[v]$, coding is simple but may incur a time complexity of $O(|V|^2)$.

- If the distance information is implemented as a priority queue, coding is more involved but the time complexity is $O(|E| \log |V|)$ when the graph is connected.
Dijkstra’s Algorithm: Example

![Graph Diagram]
#include <values.h>

// adjacency nodes

struct edge {
    int v, wt;
    struct edge *nxt;
};

// adjacency lists

int n, m;
struct edge **adj;
// priority queue with
// priority updates and
// indirect priorities

int N;
// build adjacency lists
insert( struct edge ** l, int v, int wt ) {
    struct edge *p, *q;
    p = *l;
    q = (struct edge *) calloc( 1, sizeof(struct edge) );
    q->v = v;
    q->wt = wt;
    q->nxt = p;
    *l = q;
}

show( struct edge *l ) {
    for( ; l; l = l->nxt )
        printf( " %d (%d)", l->v, l->wt );
    printf( "\n" );
}
siftdown( int i ) {
    int hi, child;

    hi = heap[i];
    while( 2*i+1 <= N-1 ) {
        child = 2*i+1;
        if(child<N-1 && dist[heap[child+1]]<dist[heap[child]])
            child++;
        if( dist[heap[child]] < dist[hi] ) {
            heap[i] = heap[child]; where[heap[i]] = i;
        } else
            break;
        i = child;
    }
    heap[i] = hi; where[heap[i]] = i;
}
heapify() {
    int i;

    for( i = N/2-1; i >= 0; i-- ) siftdown( i );
}

int heap_del() {
    int v;
    v = heap[0];
    delete[v] = 1;
    heap[0] = heap[N-1];
    where[heap[0]] = 0;
    N--;
    siftdown( 0 );
    return v;
}
heap_decrement( int v, int val ) {
    int i;
    dist[v] = val;
    i = where[v];
    while( i > 0 && val < dist[heap[(i+1)/2 - 1]] ) {
        heap[i] = heap[(i+1)/2 - 1];
        where[heap[i]] = i;
        i = (i+1)/2 - 1;
    }
    heap[i] = v;
    where[heap[i]] = i;
}
heap_show() {
    int i;

    printf("*\n");
    for( i = 0; i < N; i++ )
        printf("%d %d %d\n", i, heap[i], dist[heap[i]] );
}
dijkstra( int start ) {
    int i, minc, v, u;
    struct edge *e;

    for( i = 0; i < n; i++ ) {
        dist[i] = MAXINT;
        heap[i] = i;
        where[i] = i;
        delete[i] = 0;
    }
    N = n;

    dist[start] = 0;  parent[start] = -1;
    heapify();
for( i = 0; i < n; i++ ) {
    v = heap_del();
    minc = dist[v];
    for( e = adj[v]; e; e = e->nxt ) {
        u = e->v;
        if( !delete[u] && minc + e->wt < dist[u] ) {
            parent[u] = v;
            heap_decrement( u, minc + e->wt );
        }
    }
    heap_show();
}
// vertices are numbered from 0

main(int ac, char *av[]) {
    int i, u, v, wt;

    scanf("%d %d", &n, &m); printf("%d %d\n", n, m);

    adj = (struct edge **) calloc(n, sizeof(struct edge *));

    for(i = 0; i < m; i++) {
        scanf("%d %d %d", &u, &v, &wt);
        printf("%d %d %d\n", u, v, wt);
        insert(&adj[u], v, wt);
        insert(&adj[v], u, wt);
    }

    for(i = 0; i < n; i++) show(adj[i]);
delete = (int *) calloc( n, sizeof(int) );
parent = (int *) calloc( n, sizeof(int) );
heap = (int *) calloc( n, sizeof(int) );
dist = (int *) calloc( n, sizeof(int) );
where = (int *) calloc( n, sizeof(int) );

if( ac > 1 )
    dijkstra( atoi(av[1]) );
else
    dijkstra( 0 );

for( i = 0; i < n; i++ )
    printf( "%d %d %d\n", i, dist[i], parent[i] );
}
Dijkstra’s Algorithm: Output

5 8
0 1 1
0 3 1
0 4 9
1 4 1
1 2 5
4 3 9
4 2 9
3 2 1
4 (9) 3 (1) 1 (1)
2 (5) 4 (1) 0 (1)
3 (1) 4 (9) 1 (5)
2 (1) 4 (9) 0 (1)
2 (9) 3 (9) 1 (1) 0 (9)
NOI only
0 2 1
1 1 4
2 4 3
3 3 0
4 0 -1
Exercises

1. A complete graph is a simple graph in which any two distinct vertices are adjacent. A complete graph of $n$ vertices is denoted $K_n$. Describe the rooted ordered tree produced by a bfs and a dfs on $K_n$.

2. Code a naive Dijkstra’s algorithm to run the given example. (By naive we mean using an array instead of a priority queue to store the distance information.)

3. What is the role of $\text{where}[\ ]$ array in the given Dijkstra’s algorithm?

4. Implement Floyd’s algorithm to run the given example.