Lab 7 Solutions

Exercise 1: Write a function `fromListToMatrix`(A) which takes an adjacency list representation of a graph A and outputs an adjacency matrix representation of the same graph. Similarly write a function `fromMatrixToList`(A) which goes in the other direction.

Example solution:

```
def fromListToMatrix(A):
    ans = []
    for i in xrange(len(A)):
        ans += [[False]*len(A)]
    for i in xrange(len(A)):
        for x in A[i]:
            ans[i][x] = True
    return ans

def fromMatrixToList(A):
    ans = []
    for i in xrange(len(A)):
        ans += [[]]
    for i in xrange(len(A)):
        for j in xrange(len(A)):
            if A[i][j]:
                ans[i] += [j]
    return ans
```

Exercise 2: Write a function `connected`(A, i, j) which takes an adjacency list A and two vertices i,j, and outputs `True` if there is a path connecting i to j in the graph represented by A, and outputs `False` otherwise.

Example solution:

Using DFS

```
def visit(x, stack, visited):
    visited[x] = True
    stack += [x]

def connected(A, i, j):
    visited = [False]*len(A)
    stack = []
    visit(i, stack, visited)
    while len(stack) > 0:
```

```
        x = stack.pop()
        if x == j:
            return True
        for y in A[x]:
            if not visited[y]:
                visit(y, stack, visited)
    return False
```

Using DFS via Recursion

```
def visit(A, i, j, visited):
    if i == j:
        return True
    visited[i] = True
    for k in A[i]:
        if (not visited[k]) and visit(A, k, j, visited):
            return True
    return False

def connected(A, i, j):
    visited = [False]*len(A)
    return visit(A, i, j, visited)
```

Using BFS

```
from collections import deque

def visit(x, queue, visited):
    visited[x] = True
    queue += [x]

def connected(A, i, j):
    visited = [False]*len(A)
    queue = deque()
    visit(i, queue, visited)
    while len(queue) > 0:
        x = queue.popleft()
        if x == j:
            return True
        for y in A[x]:
            if not visited[y]:
                visit(y, queue, visited)
    return False
```

Exercise 3: Write a function `path`(A, i, j) which takes an adjacency list A and two vertices i,j, and outputs a path from i to j in the graph represented by A in `list` form, or [] if no such path exists.

For example, if we can get from vertex 0 to 5 by taking the sequence of edges $(0, 1), (1, 4), (4, 7), (7, 5)$ in some graph, then the function should return the `list` [0, 1, 4, 7, 5].

Using DFS

```
def visit(x, last, stack, visited, previous):
    visited[x] = True
    previous[x] = last
    stack += [x]

def path(A, i, j):
    visited = [False]*len(A)
    # previous[k] is the vertex right before k on a path from i to k
    previous = [-1]*len(A)
    stack = []
    visit(i, -1, stack, visited, previous)
    while len(stack) > 0:
        x = stack.pop()
        if x == j:
            ans = [j]
            while ans[len(ans)-1] != i:
                ans += [previous[ans[len(ans)-1]]]
            ans.reverse()
            return ans
        for y in A[x]:
            if not visited[y]:
                visit(y, x, stack, visited, previous)
    return []
```

Using DFS via Recursion

```
def visit(A, i, j, visited, previous):
    if i == j:
        return True
    visited[i] = True
    for k in A[i]:
        if not visited[k]:
            previous[k] = i
            if visit(A, k, j, visited, previous):
                return True
    return False

def path(A, i, j):
    visited = [False]*len(A)
    previous = [-1]*len(A)
    if not visit(A, i, j, visited, previous):
```

```
        return []
    ans = [j]
    while ans[len(ans)-1] != i:
        ans += [previous[ans[len(ans)-1]]]
    ans.reverse()
    return ans
```

Using BFS

```
from collections import deque

def visit(x, last, queue, visited, previous):
    visited[x] = True
    previous[x] = last
    queue += [x]

def path(A, i, j):
    visited = [False]*len(A)
    previous = [-1]*len(A)
    queue = deque()
    visit(i, -1, queue, visited, previous)
    while len(queue) > 0:
        x = queue.popleft()
        if x == j:
            ans = [j]
            while ans[len(ans)-1] != i:
                ans += [previous[ans[len(ans)-1]]]
            ans.reverse()
            return ans
        for y in A[x]:
            if not visited[y]:
                visit(y, x, queue, visited, previous)
    return []
```

Exercise 4: Write a function `numComponents`(A) which takes an adjacency list A and outputs the number of connected components in the graph represented by A.

Example solution:

Using DFS

```
def visit(x, stack, visited):
    visited[x] = True
    stack += [x]
```

```
def numComponents(A):
    visited = [False]*len(A)
    ans = 0
    for i in xrange(len(A)):
        if not visited[i]:
            ans += 1
            stack = []
            visit(i, stack, visited)
            while len(stack) > 0:
                x = stack.pop()
                for y in A[x]:
                    if not visited[y]:
                        visit(y, stack, visited)
    return ans
```

Using DFS via Recursion

```
def visit(A, i, visited):
    visited[i] = True
    for j in A[i]:
        if not visited[j]:
            visit(A, j, visited)

def numComponents(A):
    visited = [False]*len(A)
    ans = 0
    for i in xrange(len(A)):
        if not visited[i]:
            ans += 1
            visit(A, i, visited)
    return ans
```

Using BFS

```
from collections import deque

def visit(x, queue, visited):
    visited[x] = True
    queue += [x]

def numComponents(A):
    visited = [False]*len(A)
    ans = 0
    for i in xrange(len(A)):
        if not visited[i]:
            ans += 1
```

```
            queue = deque()
            visit(i, queue, visited)
            while len(queue) > 0:
                x = queue.popleft()
                for y in A[x]:
                    if not visited[y]:
                        visit(y, queue, visited)
    return ans
```

Exercise 5: Write a function `distance`(A, i, j) which takes an adjacency list A and two vertices i,j, and outputs the length of the shortest path from i to j in the graph represented by A. This can be done by modifying the code for bfs to keep track of distances.

Example solution:

```
from collections import deque

# D is the distance to x
def visit(x, queue, distance, D):
    distance[x] = D
    queue += [x]

# returns -1 if there's no path from i to j
# otherwise returns the distance
def distance(A, i, j):
    distance = [-1]*len(A)
    queue = deque()
    visit(i, queue, distance, 0)
    while len(queue) > 0:
        x = queue.popleft()
        if x == j:
            return distance[j]
        for y in A[x]:
            if distance[y] == -1:
                visit(y, queue, distance, distance[x] + 1)
    return -1
```