Lecture 14

Graphs: worked examples.

Rainbow ride:

(Problem source: Kurukshetra 09 OPC, see `http://www.spoj.pl/problems/RAINBOW/`). A large family of n people is going on vacation, and they are deciding who should go on a certain roller coaster ride. Each person in the family likes some other people in the family, and a person will only go on the ride if everyone he likes and everyone who likes him also goes on the ride. Each person has a weight, and the roller coaster can only support a total weight of W. Note: if two people don't like each other, they're still allowed to go on the roller coaster together. How do we choose who to go on the ride so that the maximum number of people participate?

How do we solve this problem? First we build a graph where family members are vertices, and there is an edge (u, v) between two family members if either u likes v or v likes u. Then if one family member goes on the ride, everyone from his connected component must go on the ride with him. So, essentially we must choose which connected components go on the ride so that the total weight of all connected components we choose is at most W. We can first find the connected components and their total sizes and weights by either DFS or BFS. Then this becomes exactly the knapsack problem from Exercise 5, Lab 6. We have some number m of items (each item is a connected component of the family graph), with item i having value v_i (the number of people in that component) and weight w_i (the total weight of the component). We must choose which items to put in our knapsack of weight W (i.e. the roller coaster ride) so as to maximize the total value. This can be solved in time $\Theta(mW)$ using memoization; see the Lab 6 solutions for details. **todo:** *fix this reference?*

Perimeter:

You are given a 2D-image drawn using 10 colors. Each color is represented by an integer from 0 to 9. The image is described by giving the color of each pixel in the image. Two pixels are adjacent if one is immediately to the left of the other or immediately above the other. The image consists of many objects, and two pixels are part of the same object if they are adjacent and have the same color. The *perimeter* of an object is the number of pixels on the boundary of the object; that is, the number of pixels either bordering another object, or bordering one of the four edges of the image. Given a pixel in an image, calculate the perimeter of the object it lies in.

Example solution: This problem can be solved using any one of depth-first search or breadth-first search. Create a graph where pixels are vertices, and two vertices have an edge between them if the pixels are adjacent. Then, we must explore the connected component of the starting pixel and add one for each vertex we encounter which is either on the edge or is adjacent to a vertex which is a part of a different object.

```
def isBorder(image, x, y):
    # check if (x,y) is on one of the four borders of the image
    if x==0 or x+1==len(image) or y==0 or y+1==len(image[0]):
        return 1
```

```
    # check if (x,y) borders a pixel with a different color
    dx = [0, 0, 1, -1]
    dy = [1, -1, 0, 0]
    for i in xrange(4):
        nx = x + dx[i]
        ny = y + dy[i]
        if image[x][y]!=image[nx][ny]:
            return 1

    # (x,y) is not on the border
    return 0

def recurse(image, x, y, seen):
    ans = isBorder(image, x, y)
    seen[x][y] = True
    dx = [0, 0, 1, -1]
    dy = [1, -1, 0, 0]
    for i in xrange(4):
        nx = x + dx[i]
        ny = y + dy[i]
        if nx>=0 and nx<len(image) and ny>=0 and ny<len(image[0]):
            if image[nx][ny]==image[x][y] and not seen[nx][ny]:
                ans += recurse(image, nx, ny, seen)
    return ans

# recursive DFS implementation
# image is a list of lists, giving pixel colors
# we should compute the answer for the object containing pixel (x,y)
def perimeter(image, x, y):
    seen = []
    for i in xrange(len(image)):
        seen += [[False]*len(image[0])]
    return recurse(image, x, y, seen)
```

Here is another implementation which uses an explicit stack. The `isBorder` function is the same as last time, so we don't repeat it here.

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

# DFS implementation with explicit stack
# image is a list of lists, giving pixel colors
# we should compute the answer for the object containing pixel (x,y)
def perimeter(image, x, y):
```

```
    seen = []
    for i in xrange(len(image)):
        seen += [[False]*len(image[0])]
    ans = 0
    stack = []
    visit(x, y, stack, seen)
    while len(stack) > 0:
        p = stack.pop()
        a = p[0]
        b = p[1]
        ans += isBorder(image, a, b)
        dx = [0, 0, 1, -1]
        dy = [1, -1, 0, 0]
        for i in xrange(4):
            nx = a + dx[i]
            ny = b + dy[i]
            if nx>=0 and nx<len(image) and ny>=0 and ny<len(image[0]):
                if image[nx][ny]==image[a][b] and not seen[nx][ny]:
                    visit(nx, ny, stack, seen)
    return ans
```

Stepping on Nails: You're in a 2D-room which is n by m meters. The room is divided into squares that are each 1 square meter each. The bottom-left square is (0,0), and the top-right is (n-1,m-1). You start in square (x,y) and are trying to get to square (u,v). From each square you can go to 8 other squares: up, down, left, right, and also the four diagonals. Some squares have nails on the floor, and if you step on it, it hurts! You don't mind stepping on a few nails, but if you step on more than N nails then you'll need to go to the hospital. Calculate the shortest way of getting to (u,v) without going to the hospital. If it is impossible, return -1. The starting and ending locations will not have nails.

Example solution: We create a graph where each vertex represents (x,y,t): the square (x, y) we are at, and the number of nails t we have stepped on so far (we only have to consider $t \leq N$). Each vertex has eight edges leading to the adjacent vertices, where t either stays the same or increases by 1, depending on whether we stepped onto a nail. We should return the minimum distance to (u, v, t) over all $0 \leq t \leq N$, which we can find using BFS. If we never managed to reach such a (u, v, t), the answer is -1.

```
from collections import deque
def visit(x, y, t, queue, distance, D):
    distance[x][y][t] = D
    queue += [[x,y,t]]

# room[i][j] is True if there's a nail and False otherwise
def fastestRoute(room, x, y, u, v, N):
    # the makeArray function from Lecture 16
```

```
    distance = makeArray([room, room[0], N+1], -1)
    queue = deque()
    visit(x, y, 0, queue, distance, 0)
    while len(queue) > 0:
        p = queue.popleft()
        a = p[0]
        b = p[1]
        t = p[2]
        if a==u and b==v:
            return distance[a][b][t]
        for dx in xrange(-1, 2):
            for dy in xrange(-1, 2):
                if dx==0 and dy==0: continue
                nx = a + dx
                ny = b + dy
                if nx>=0 and nx<len(room) and ny>=0 and ny<len(room[0]):
                    nt = t
                    if room[nx][ny]: nt += 1
                    if nt <= N:
                        visit(nx, ny, nt, queue, distance, distance[a][b][t]+1)
    return -1
```

Catching a cockroach: You and a cockroach are in a room. You hate cockroaches, so you want to catch it and kill it. You start at location (x,y) and the cockroach starts at (u,v). The room is n x m meters, and the bottom-left square is (0,0), and the top-right is (n-1,m-1). Cockroaches are simple creatures, and you know how the cockroach behaves: you are given two lists dx and dy of moves so that you know that in one step, the cockroach moves to (u + dx[0], v + dy[0]), then from there moves to (u + dx[0] + dx[1], v + dy[0], v + dy[1]), etc. In other words, the cockroach moves by (dx[i], dy[i]) in the ith step. If any particular step would make the cockroach run into a wall, the cockroach just stays where he is in that step. If he reaches the end of his list of moves, he cycles back and does the 0th move again. What is the minimum amount of time required to catch the cockroach? If any step, we can move either up, down, left, or right. We can also just choose to stay where we are.

Example solution: We create a graph where the vertices represent (a,b,c,d,i), where our location is (a,b), the cockroach's location is (c,d), and the cockroach's next move is (dx[i], dy[i]). We need to return the minimum distance to any vertex where (a,b) equals (c,d), and i can be any integer $0 \leq i < \text{len}(dx)$. This can be done using breadth first search.

```
from collections import deque
def visit(x, y, u, v, i, queue, distance, D):
    distance[x][y][u][v] = D
    queue += [[x,y,u,v,i]]

# The room is only n x m meters
```

```
def catchRoach(x, y, u, v, n, m, dx, dy):
    # the makeArray function from Lecture 16
    distance = makeArray([n, m, n, m, len(dx)], -1)
    queue = deque()
    visit(x, y, u, v, 0, queue, distance, 0)
    while len(queue) > 0:
        p = queue.popleft()
        a = p[0]
        b = p[1]
        c = p[2]
        d = p[3]
        i = p[4]
        if a==c and b==d:
            return distance[a][b][c][d][i]
        cx = [0, 0, -1, 1, 0]
        cy = [-1, 1, 0, 0, 0]
        # calculate the roach's new location
        nrx = c + dx[i]
        nry = d + dy[i]
        # the roach can't leave the room
        if nrx<0 or nrx==n or nry<0 or nry==m:
            nrx = c
            nry = d
        for j in xrange(5):
            nx = a + cx[j]
            ny = b + cy[j]
            if nx>=0 and nx<n and ny>=0 and ny<m:
                visit(nx, ny, nrx, nry, (i+1)%len(dx),
                      queue, distance, distance[a][b][c][d][i]+1)
```