Grafos implícitos

Recorrer sin construir el grafo

Carlos A Delgado S

Pontificia Universidad Javeriana Cali

Septiembre de 2026

Objetivos de la sesión

Al terminar la clase, usted podrá

- Justificar que los dos recorridos cuestan $\Theta(V + E)$ sobre lista de adyacencia, y decir qué cambia sobre las otras representaciones.
- Reconocer un grafo implícito: uno cuyos vértices y aristas se calculan con una regla en vez de guardarse.
- Recorrer una cuadrícula con DFS y con BFS sin construir la lista de adyacencia, y decir qué cuesta.
- Llevar un problema de juez a un grafo: decidir qué es un vértice, qué es una arista y qué recorrido contesta la pregunta.
- Reconstruir la ruta que la amplitud encontró y contar el componente más grande de un grafo.

Contenido

1 Recuento

2 Grafos implícitos: Jaimico y el café

3 UVa 10977: el bosque encantado

4 UVa 627: la ruta con menos saltos

5 UVa 11749: el componente más grande

6 Errores comunes

7 Ejercicios

8 Referencias

Contenido

1 Recuento

2 Grafos implícitos: Jaimico y el café

3 UVa 10977: el bosque encantado

4 UVa 627: la ruta con menos saltos

5 UVa 11749: el componente más grande

6 Errores comunes

7 Ejercicios

8 Referencias

Los dos recorridos

DFSAux(v)

1. Marcar v como visitado
2. Para cada nodo u adyacente a v:
   2.1 Si u no ha sido visitado ejecutar *DFSAux*(u)

Los dos recorridos

$BFS(G, s)$

1. Hacer $s.distancia = 0$ y agregar s a la cola P
2. Mientras la cola P no esté vacía:
    2.1 Retirar el frente w de la cola P
    2.2 Para cada nodo u adyacente a w:
        2.2.1 Si $u.distancia = \infty$: hacer $u.distancia = w.distancia + 1$ y agregar u a P

Lo que comparten

Los dos visitan exactamente los vértices alcanzables desde el arranque, cada uno una sola vez, y miran cada arista una vez por cada extremo. La profundidad marca al entrar; la amplitud, al encolar, y deja la distancia en aristas.

Lo que cuestan

Teorema

Recorrer un grafo $G = (V, E)$ en profundidad o en amplitud, guardado como lista de adyacencia, cuesta $\Theta(V + E)$.

Demostración

Se procede contando por separado lo que aporta cada vértice y lo que aporta cada arista. Cada vértice se marca una sola vez, porque antes de entrar se pregunta por la marca: $\Theta(V)$ en total. Estando en u se recorre su lista de vecinos, de $\delta(u)$ entradas, y como u se visita una vez esa lista se recorre una vez; la suma sobre todos los vértices es $\sum_u \delta(u)$, que vale $2|E|$ en el grafo no dirigido y $|E|$ en el dirigido: $\Theta(E)$. Sumando, $\Theta(V + E)$.

Lo que cuestan

Conclusión

Por lo tanto, se puede concluir que ambos recorridos cuestan $\Theta(V + E)$ sobre lista de adyacencia. Sobre matriz de adyacencia cuestan $\Theta(V^2)$, porque los vecinos de u obligan a recorrer su fila completa; sobre lista de aristas, $\Theta(V \cdot E)$.

Contenido

1 Recuento

2 Grafos implícitos: Jaimico y el café

3 UVa 10977: el bosque encantado

4 UVa 627: la ruta con menos saltos

5 UVa 11749: el componente más grande

6 Errores comunes

7 Ejercicios

8 Referencias

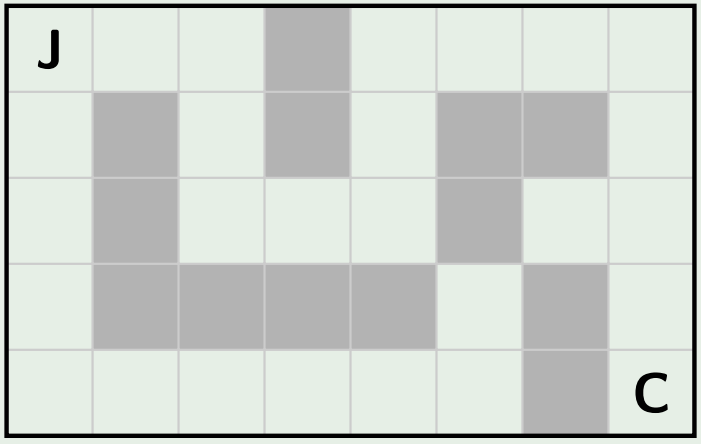
Jaimico y el café

El problema

Un laberinto de n filas y m columnas guardado en una matriz M: $M[i][j]$ vale 0 si la celda está libre, 1 donde está Jaimico, 2 si es pared y 3 donde está el café. En cada paso Jaimico se mueve a una celda contigua que no sea pared: arriba, abajo, izquierda o derecha. ¿Puede llegar al café? Y si puede, ¿en cuántos pasos como mínimo?

Jaimico y el café

El laberinto de hoy



$n = 5$, $m = 8$. **J** es Jaimico ($M[0][0] = 1$), **C** el café ($M[4][7] = 3$) y las celdas grises valen 2.

Jaimico y el café

¿Entendimos el problema?

Entrada: la matriz M. **Salida:** si hay camino, y el número mínimo de pasos. Con el dedo: el corredor de abajo parece la ruta obvia y termina en un callejón; la que llega baja dos filas, cruza por debajo de la pared, vuelve a subir y sigue por arriba hasta la última columna. Contar esos pasos a mano ya es un recorrido.

La cuadrícula es un grafo

Definición (Grafo implícito)

Un grafo cuyos vértices y aristas no están guardados en ninguna estructura: hay una regla que dice qué es un vértice y una regla que, dado un vértice, produce sus vecinos. El recorrido llama a esa regla cada vez que necesita la lista de adyacencia de alguien.

En el laberinto

- **Vértices:** las celdas (r, c) con $M[r][c] \neq 2$.
- **Aristas:** $\{(r, c), (r', c')\}$ cuando las dos celdas no son pared y son contiguas: $|r - r'| + |c - c'| = 1$.
- **Vecinos de** (r, c)**:** a lo sumo cuatro, y se obtienen sumando los cuatro desplazamientos y descartando lo que se sale de la matriz o cae en pared.

La cuadrícula es un grafo

Por qué no se construye

La lista de adyacencia de un laberinto de 500×500 tendría $250\,000$ listas y hasta un millón de entradas, todas deducibles de una suma. Se gastaría $\Theta(n \cdot m)$ en memoria y en tiempo solo para escribir lo que la regla ya dice.

El primer intento: un `if` por dirección

```
def dfsAuxIngenuo(rJ, cJ, rC, cC, M, vis):
    # Primer intento: un if por direccion
    vis[rJ][cJ] = True
    ans = rJ == rC and cJ == cC
    if not ans and rJ - 1 >= 0 and not vis[rJ - 1][cJ] and M[rJ - 1][cJ] != 2:
        ans = dfsAuxIngenuo(rJ - 1, cJ, rC, cC, M, vis)
    if not ans and rJ + 1 < len(M) and not vis[rJ + 1][cJ] and M[rJ + 1][cJ]
    ↪  != 2:
        ans = dfsAuxIngenuo(rJ + 1, cJ, rC, cC, M, vis)
    if not ans and cJ - 1 >= 0 and not vis[rJ][cJ - 1] and M[rJ][cJ - 1] != 2:
        ans = dfsAuxIngenuo(rJ, cJ - 1, rC, cC, M, vis)
    if not ans and cJ + 1 < len(M[rJ]) and not vis[rJ][cJ + 1] and M[rJ][cJ +
    ↪  1] != 2:
        ans = dfsAuxIngenuo(rJ, cJ + 1, rC, cC, M, vis)
    return ans
```

El primer intento: un `if` por dirección

Funciona, y cuesta mantenerlo

Cuatro veces la misma comprobación con los números cambiados. Para permitir diagonales habría que escribir cuatro `if` más, y para el caballo de ajedrez, ocho. El movimiento está repartido por todo el código en vez de estar en un solo lugar.

La regla de los vecinos en una tabla

```
dr = [0, -1, 0, 1]
dc = [-1, 0, 1, 0]

def dfsAux(rJ, cJ, rC, cC, M, vis):
    # True si desde (rJ, cJ) se llega al cafe; se detiene al
    ↪  encontrarlo
    vis[rJ][cJ] = True
    ans = rJ == rC and cJ == cC
    i = 0
    while i < 4 and not ans:
        nr = rJ + dr[i]
        nc = cJ + dc[i]
        if nr >= 0 and nr < len(M) and nc >= 0 and nc < len(M[nr]):
            if not vis[nr][nc] and M[nr][nc] != 2:
                ans = dfsAux(nr, nc, rC, cC, M, vis)
        i = i + 1
    return ans
```

La regla de los vecinos en una tabla

Lo que cambió

`dr[i]` y `dc[i]` son el *i*-ésimo movimiento: izquierda, arriba, derecha, abajo. Las cuatro comprobaciones se volvieron una dentro del `while`, y la condición `and not ans` detiene la búsqueda en cuanto el café aparece. Es el *DFSAux* de siempre con la lista de adyacencia reemplazada por una suma.

¿Puede llegar?

```
def dfs(M):
    # Puede Jaimico llegar al cafe?
    n = len(M)
    m = len(M[0])
    jaimico = posicion_de(M, 1)
    cafe = posicion_de(M, 3)
    vis = tabla(n, m, False)
    return dfsAux(jaimico[0], jaimico[1], cafe[0], cafe[1], M, vis)
```

Las dos ayudantes

`posicion_de(M, valor)` recorre la matriz y devuelve la fila y la columna donde está el valor; `tabla(n, m, valor)` arma una matriz de $n \times m$ llena de ese valor, creando una lista nueva por fila. La marca es una matriz porque el vértice tiene dos coordenadas.

¿En cuántos pasos? Amplitud

```
def bfs(M):
    # Pasos minimos de Jaimico al cafe; -1 si no se llega
    n = len(M)
    m = len(M[0])
    jaimico = posicion_de(M, 1)
    cafe = posicion_de(M, 3)
    d = tabla(n, m, -1)
    d[jaimico[0]][jaimico[1]] = 0
    cola = deque()
    cola.append(jaimico)
    while len(cola) > 0:
        actual = cola.popleft()
        r = actual[0]
        c = actual[1]
        i = 0
        while i < 4:
            nr = r + dr[i]
            nc = c + dc[i]
            if nr >= 0 and nr < n and nc >= 0 and nc < m:
                if M[nr][nc] != 2 and d[nr][nc] == -1:
                    d[nr][nc] = d[r][c] + 1
                    cola.append((nr, nc))
            i = i + 1
    return d[cafe[0]][cafe[1]]
```

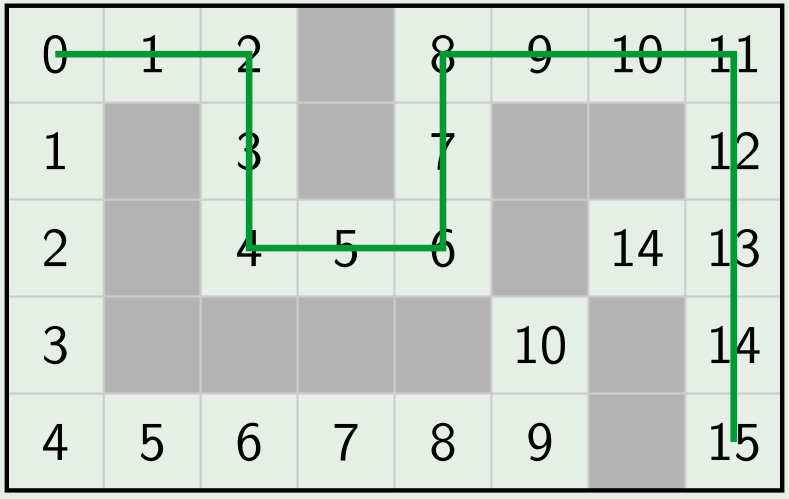
¿En cuántos pasos? Amplitud

La distancia hace de marca

Igual que sobre la lista de adyacencia: -1 es "no visto", y la distancia se fija al encolar. La amplitud no se detiene al encontrar el café: sigue hasta vaciar la cola, y al final `d` tiene la distancia a todas las celdas alcanzables.

¿En cuántos pasos? Amplitud

`bfs(M)` sobre el laberinto de hoy



El café queda a 15 pasos por el camino verde. El corredor de abajo llega hasta la celda marcada 10 y ahí se cierra: la amplitud lo explora igual y lo descarta sola.

Lo que cuesta

$\Theta(n \cdot m)$

El grafo tiene a lo sumo $n \cdot m$ vértices y cada uno tiene a lo sumo cuatro aristas, así que $E \leq 2nm$. El $\Theta(V + E)$ de los recorridos se convierte en $\Theta(n \cdot m)$, y calcular los vecinos cuesta $\Theta(1)$ por celda: la regla no agrega nada al orden. El Teorema 22.5 vale sin cambios: nada del argumento dependía de que el grafo estuviera guardado.

La recursión sobre una cuadrícula grande

Un laberinto de 500×500 con un corredor largo lleva la profundidad recursiva a 250 000 llamadas anidadas, y Python se detiene mucho antes de eso. Sobre cuadrículas grandes, la amplitud con cola o la profundidad con pila explícita son las versiones que corren.

Contenido

1 Recuento

2 Grafos implícitos: Jaimico y el café

3 UVa 10977: el bosque encantado

4 UVa 627: la ruta con menos saltos

5 UVa 11749: el componente más grande

6 Errores comunes

7 Ejercicios

8 Referencias

Enchanted Forest

UVa 10977 — Enchanted Forest

Enunciado: `https://vjudge.net/problem/UVA-10977`
Envío: `https://vjudge.net/problem/UVA-10977`

El problema

Un bosque de $R \times C$ intersecciones. Se entra por $(1, 1)$ y se sale por (R, C), moviendo una unidad por paso. Hay m posiciones bloqueadas y n Jigglypuffs; cada uno, en (x, y) con volumen L, hace peligrosa toda posición a distancia L o menos de él. Hay que dar la longitud del camino más corto que evite lo bloqueado y lo peligroso, o `Impossible.` si no existe.

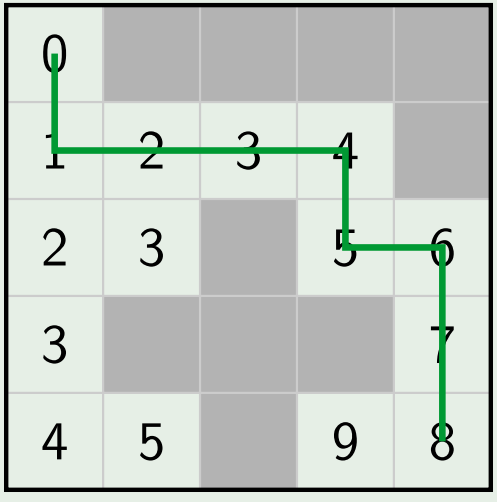
Enchanted Forest

Entrada y salida

Entrada: varios casos; cada uno con R y C ($1 \leq R, C \leq 200$), luego m y m pares, luego n ($0 \leq n \leq 100$) y n triplas x, y, L ($1 \leq L \leq 100$). Termina con $R = C = 0$. **Salida:** una línea por caso.

¿Entendimos el problema?

La muestra: 5×5, cinco bloqueadas y un Jigglypuff en $(4, 3)$ con $L = 1$.
Respuesta: 8



Gris: las cuatro bloqueadas de la primera fila, la de $(2, 5)$, y las cinco que el Jigglypuff vuelve peligrosas: él mismo y sus cuatro vecinas a distancia 1. Cada celda lleva su distancia a $(1, 1)$.

¿Entendimos el problema?

Qué cambia respecto a Jaimico

Nada en el recorrido. Lo nuevo es **cómo se arma la matriz**: las celdas peligrosas no vienen listadas, se calculan a partir de cada Jigglypuff con la distancia euclidiana $(x - i)^2 + (y - j)^2 \leq L^2$.

Armar el mundo

```
def marcar_peligro(mundo, R, C, x, y, L):
    # Bloquea las celdas a distancia euclidiana L o menos de (x, y)
    i = max(1, x - L)
    while i <= min(R, x + L):
        j = max(1, y - L)
        while j <= min(C, y + L):
            if (x - i) * (x - i) + (y - j) * (y - j) <= L * L:
                mundo[i][j] = -1
            j = j + 1
        i = i + 1
```

Solo el cuadrado que rodea al Jigglypuff

Recorrer todo el mundo por cada Jigglypuff costaría $n \cdot R \cdot C$. Basta mirar el cuadrado de lado $2L+1$ centrado en él, recortado al tablero, y quedarse con lo que cae dentro del círculo. Comparar con L^2 evita la raíz cuadrada.

Armar el mundo

Las coordenadas del enunciado

Aquí las filas y columnas van de 1 a R y de 1 a C, como en el enunciado; la fila y la columna 0 de `mundo` no se usan. Es la otra forma de resolver el desfase: en lugar de restar 1 al leer, se reserva la posición 0.

Salir del bosque

```
def salida_mas_corta(mundo, R, C):
    # Pasos minimos de (1, 1) a (R, C) por celdas con 0; -1 si no hay
    d = matriz(R, C, -1)
    cola = deque()
    if mundo[1][1] == 0:
        d[1][1] = 0
        cola.append((1, 1))
    while len(cola) > 0:
        actual = cola.popleft()
        r = actual[0]
        c = actual[1]
        k = 0
        while k < 4:
            nr = r + dr[k]
            nc = c + dc[k]
            if nr >= 1 and nr <= R and nc >= 1 and nc <= C:
                if mundo[nr][nc] == 0 and d[nr][nc] == -1:
                    d[nr][nc] = d[r][c] + 1
                    cola.append((nr, nc))
            k = k + 1
    return d[R][C]
```

Salir del bosque

El caso límite que se escapa

Si la propia $(1, 1)$ es peligrosa, no se encola nada y la cola arranca vacía: `d[R][C]` queda en -1 y se imprime `Impossible.` sin ningún caso aparte.

Costo

Armar el mundo cuesta $\Theta(R \cdot C + n \cdot L^2)$ y el recorrido $\Theta(R \cdot C)$. Con $R = C = 200$ y cien Jigglypuffs de volumen 100 son unos 10^6 más $4 \cdot 10^4$: cabe de sobra.

Contenido

1 Recuento

2 Grafos implícitos: Jaimico y el café

3 UVa 10977: el bosque encantado

4 UVa 627: la ruta con menos saltos

5 UVa 11749: el componente más grande

6 Errores comunes

7 Ejercicios

8 Referencias

The Net

UVa 627 — The Net

Enunciado: `https://vjudge.net/problem/UVA-627`
Envío: `https://vjudge.net/problem/UVA-627`

El problema

Una red de a lo sumo 300 enrutadores. Cada uno tiene su lista de enrutadores *visibles*, en orden ascendente. Para cada consulta —un origen y un destino— hay que imprimir la ruta con menos saltos, y si hay varias con los mismos saltos, la de identificadores más bajos; si no hay ruta, `connection impossible`.

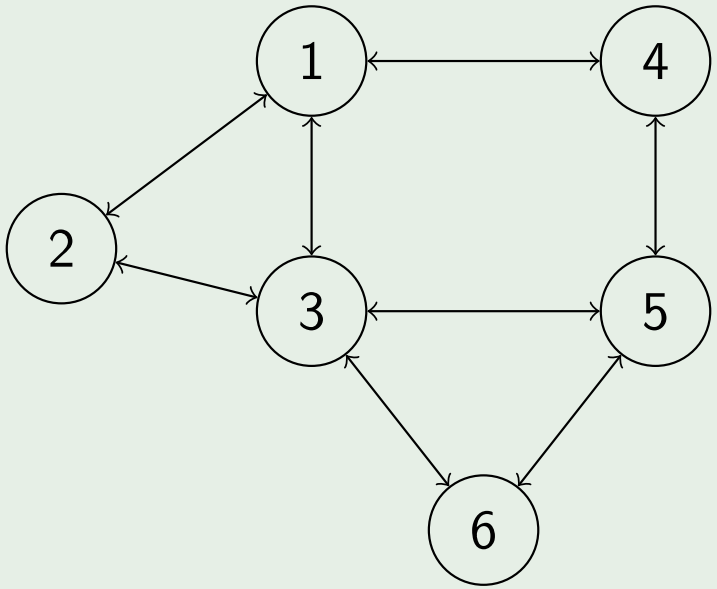
The Net

Entrada y salida

Entrada: varias redes. Cada una trae n, luego n líneas de la forma `3-1,2,5,6` —el enrutador y sus visibles—, luego el número de consultas y una línea por consulta con origen y destino. **Salida:** por red, una línea con cinco guiones y después una línea por consulta.

¿Entendimos el problema?

La primera red de la muestra



`1-2,3,4`, `2-1,3`, `3-1,2,5,6`, `4-1,5`, `5-3,4,6`, `6-3,5`. La consulta `1 6` responde `1 3 6`: dos saltos, y no hay otra de dos saltos.

¿Entendimos el problema?

Las visibles no tienen por qué ser simétricas

En la segunda red, `5-1` pero `1-2`: el 5 ve al 1 y el 1 no ve al 5. El grafo es **dirigido**, y la consulta `5 9` da `connection impossible` aunque el 9 sí vea al 5.

El grafo y el algoritmo

Qué es qué

- **Vértices:** los enrutadores. **Aristas:** de u a cada visible de u, dirigidas.
- *Menos saltos* es la distancia en aristas: amplitud desde el origen.
- La ruta se necesita completa, no solo su largo: cada enrutador guarda desde cuál se llegó a él, y la ruta se reconstruye subiendo por esos predecesores.

El grafo y el algoritmo

$Ruta(G, s, t)$

1. Hacer $s.pred = NIL$ y agregar s a la cola P
2. Mientras P no esté vacía y t no tenga predecesor:
    2.1 Retirar el frente w de P
    2.2 Para cada u visible desde w, en orden ascendente:
        2.2.1 Si u no tiene predecesor: hacer $u.pred = w$ y agregar u a P
3. Si t no tiene predecesor, no hay ruta; si no, subir por los predecesores desde t hasta s e invertir

El grafo y el algoritmo

De dónde sale "la de identificadores más bajos"

Las listas vienen ascendentes y la primera vez que se descubre un enrutador queda fijado su predecesor. Por capas, los enrutadores salen de la cola en el orden de sus rutas, así que la primera ruta que llega a cada uno es la menor de su largo.

El código

```
def ruta(grafo, inicio, destino):
    # Lista de enrutadores de inicio a destino, o [] si no hay conexion
    pred = {}
    pred[inicio] = -1
    cola = deque()
    cola.append(inicio)
    while len(cola) > 0 and destino not in pred:
        u = cola.popleft()
        for v in grafo[u]:
            if v not in pred:
                pred[v] = u
                cola.append(v)
    camino = []
    if destino in pred:
        v = destino
        while v != -1:
            camino.append(v)
            v = pred[v]
        camino.reverse()
    return camino
```

El código

Dos detalles del código

`pred` es un diccionario que hace de marca y de predecesor a la vez: estar en él es haber sido descubierto. Y la condición del `while` detiene la amplitud en cuanto el destino tiene predecesor: lo que falta por explorar no puede dar una ruta más corta.

Costo

Por consulta, una amplitud: $\Theta(V + E)$ con $V \leq 300$ y $E \leq 300 \cdot 50$. Leer la red es lo más delicado: partir cada línea por el guión y después por las comas, y aceptar la lista vacía de 2-.

Contenido

1 Recuento

2 Grafos implícitos: Jaimico y el café

3 UVa 10977: el bosque encantado

4 UVa 627: la ruta con menos saltos

5 UVa 11749: el componente más grande

6 Errores comunes

7 Ejercicios

8 Referencias

Poor Trade Advisor

UVa 11749 — Poor Trade Advisor

Enunciado: `https://vjudge.net/problem/UVA-11749`
Envío: `https://vjudge.net/problem/UVA-11749`

El problema

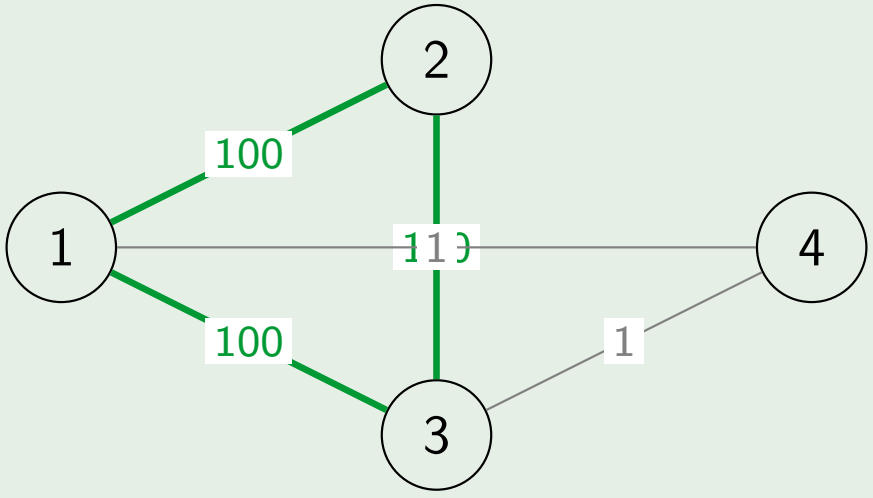
n ciudades y m carreteras bidireccionales, cada una con una ganancia anual (PPA). Una provincia es un conjunto conexo y no vacío de carreteras con sus ciudades, y tiene que tener la mayor PPA promedio posible. ¿Cuántas ciudades puede tener, como máximo, una provincia así?

Entrada y salida

Entrada: varios casos; cada uno con n ($1 < n \leq 500$) y m ($1 \leq m \leq 10^6$), y m líneas con los dos extremos y la PPA. Termina con $n = m = 0$. **Salida:** una línea por caso.

¿Entendimos el problema?

El primer caso de la muestra: respuesta 3



El promedio más alto posible es 100, y solo se consigue con carreteras de PPA 100: meter una de PPA 1 baja el promedio. Con las de 100 quedan conectadas 1, 2 y 3: tres ciudades.

¿Entendimos el problema?

La idea que resuelve el problema

El promedio de un conjunto nunca supera a su elemento más grande, y lo iguala solo cuando todos los elementos son ese máximo. Así que la provincia usa únicamente carreteras de PPA máxima, y la pregunta se vuelve: en el grafo con solo esas carreteras, ¿cuál es el componente conexo más grande?

El código

```
def ppa_maxima(aristas):
    # La mayor PPA entre todas las carreteras
    mayor = aristas[0][2]
    for arista in aristas:
        if arista[2] > mayor:
            mayor = arista[2]
    return mayor


def grafo_de_las_mejores(n, aristas, mayor):
    # Lista de adyacencia con solo las carreteras de PPA igual a mayor
    adj = []
    i = 0
    while i <= n:
        adj.append([])
        i = i + 1
    for arista in aristas:
        if arista[2] == mayor:
            adj[arista[0]].append(arista[1])
            adj[arista[1]].append(arista[0])
    return adj
```

El código

```
def provincia_mas_grande(n, aristas):
    # aristas: lista de (u, v, ppa); el componente mas grande del
    ↪  grafo
    # que queda con solo las carreteras de PPA maxima
    adj = grafo_de_las_mejores(n, aristas, ppa_maxima(aristas))
    visitado = [False] * (n + 1)
    mejor = 0
    u = 1
    while u <= n:
        if not visitado[u]:
            t = tamano_desde(adj, u, visitado)
            if t > mejor:
                mejor = t
        u = u + 1
    return mejor
```

El código

```
def tamano_desde(adj, s, visitado):
    # Cuenta las ciudades del componente de s
    total = 0
    visitado[s] = True
    pila = [s]
    while len(pila) > 0:
        u = pila.pop()
        total = total + 1
        for v in adj[u]:
            if not visitado[v]:
                visitado[v] = True
                pila.append(v)
    return total
```

El código

```
def tamano_desde(adj, u, visitado):
    # La misma cuenta, con la pila de llamadas en lugar de la lista
    visitado[u] = True
    total = 1
    for v in adj[u]:
        if not visitado[v]:
            total = total + tamano_desde(adj, v, visitado)
    return total
```

Las dos son el mismo recorrido

Es el *DFSAux* de los recorridos con un contador: cada llamado devuelve cuántas ciudades tocó y el de `u` suma las de sus vecinos nuevos más él mismo. Lo que antes guardaba `pila` lo guarda ahora la pila de llamadas: cada llamado pendiente es un vértice apilado.

Una cadena de n ciudades apila n llamados y Python corta cerca de los mil; con la lista no hay tope. La recursiva se escribe primero, porque es la que ya se sabe; al juez, con n grande, va la de la lista.

El código

Es el ciclo externo de siempre

El `while` sobre `u` es el $DFS(G)$ completo: cada vez que encuentra una ciudad sin marcar, arranca un recorrido nuevo, y ese recorrido es exactamente un componente. Lo único que se agrega es contar lo que el recorrido toca y quedarse con el mayor.

Costo

Hallar el máximo y armar el grafo, $\Theta(m)$; el recorrido completo, $\Theta(n+m)$. Con m hasta un millón lo que pesa es leer la entrada: se lee todo de una vez y se parte por espacios.

Contenido

1 Recuento

2 Grafos implícitos: Jaimico y el café

3 UVa 10977: el bosque encantado

4 UVa 627: la ruta con menos saltos

5 UVa 11749: el componente más grande

6 Errores comunes

7 Ejercicios

8 Referencias

Los que más cuestan

Sobre cuadrículas

- Salirse del tablero: la regla revisa el rango antes de mirar la celda. En Python un índice negativo no falla, lee desde el final, y el error queda escondido.
- Confundir fila con columna al leer, o la numeración desde 1 del enunciado con la desde 0 del programa.
- Usar profundidad recursiva sobre una cuadrícula grande.
- Buscar la distancia mínima con profundidad: llega, pero no por el camino corto.

Los que más cuestan

Sobre las rutas y los componentes

- Tratar como no dirigido un grafo que el enunciado da dirigido, como las listas de visibles de The Net.
- Reconstruir la ruta sin invertirla: sale del destino al origen.
- Olvidar el ciclo externo cuando la pregunta es por todos los componentes.
- Leer un millón de líneas con `input()`: se lee todo de una vez.

Contenido

1 Recuento

2 Grafos implícitos: Jaimico y el café

3 UVa 10977: el bosque encantado

4 UVa 627: la ruta con menos saltos

5 UVa 11749: el componente más grande

6 Errores comunes

7 Ejercicios

8 Referencias

Rumor

Codeforces 893C — Rumor

`https://codeforces.com/problemset/problem/893/C`

El problema

En una ciudad hay n personajes; algunos pares son amigos. Sobornar al personaje i cuesta c_i monedas y hace que empiece a contar un rumor; quien lo oye se lo cuenta a todos sus amigos, gratis. ¿Cuál es el mínimo de monedas para que todos los n conozcan el rumor?

Entrada y salida

Varios casos, cada uno con n y m ($1 \leq n \leq 2 \cdot 10^4$, $0 \leq m \leq 4 \cdot 10^4$), la línea con $c_1, \ldots, c_n$ y m pares de amigos numerados desde 1; termina con $n = m = 0$. Una línea por caso. Con $n = 5$, costos $2, 5, 3, 4, 8$ y amigos $(1, 4)$, $(4, 5)$ la respuesta es 10.

Rumor

Cómo atacarlo

Es el problema de la provincia al revés: aquí se necesitan todos los componentes, y de cada uno importa el mínimo en vez del tamaño.

Robot

El problema

Un robot patrulla una cuadrícula de $m \times n$ ($1 \leq m, n \leq 100$) y se mueve a una celda contigua por paso. Algunas celdas tienen obstáculos: entrar en una obliga a usar el turbo, y el robot no puede pasar por más de k celdas con obstáculo seguidas ($0 \leq k \leq 100$). Hay que hallar el mínimo de pasos de $(1, 1)$ a (m, n), que son celdas libres, o -1 si no se puede.

Entrada y salida

El número de casos; por caso, m y n, luego k, y m filas de n enteros: 1 es obstáculo, 0 es libre. Una línea por caso. Con $m = 2$, $n = 5$, $k = 0$ y las filas `0 1 0 0 0` y `0 0 0 1 0` la respuesta es 7.

Robot

Cómo atacarlo

Con $k = 1$ basta llevar la racha en la celda. Con $k \geq 2$ no: a una misma celda se puede llegar con rachas distintas, y la marca tiene que distinguirlas. Piense qué es el vértice antes de escribir la cola.

Tres más para el juez

UVa 439 — Knight Moves

Enunciado: `https://vjudge.net/problem/UVA-439`
Envío: `https://vjudge.net/problem/UVA-439`
Las 64 casillas y los ocho saltos del caballo: `dr` y `dc` con ocho entradas y la amplitud de hoy.

UVa 469 — Wetlands of Florida

Enunciado: `https://vjudge.net/problem/UVA-469`
Envío: `https://vjudge.net/problem/UVA-469`
El tamaño del lago que contiene a una celda es el tamaño de su componente. La trampa está en la lectura de la entrada.

Tres más para el juez

UVa 10653 — Bombs! NO they are Mines!!

Enunciado: `https://vjudge.net/problem/UVA-10653`
Envío: `https://vjudge.net/problem/UVA-10653`
El laberinto de Jaimico con las minas descritas por filas. Con 1000×1000 celdas la amplitud cabe y la profundidad recursiva no.

Para practicar en papel

Sobre el laberinto de hoy

1. Corra `dfsAux` desde Jaimico siguiendo el orden de `dr`, `dc` y escriba el orden en que se visitan las primeras diez celdas. Compare con el orden de la amplitud.
2. Si Jaimico también pudiera moverse en diagonal, ¿cuántas entradas tendrían `dr` y `dc` y en cuánto quedaría la distancia al café?
3. Dibuje un laberinto de 4×4 donde la profundidad llegue al café por un camino de más del doble de la distancia mínima.

Para practicar en papel

Sobre los tres problemas

1. En The Net, ¿qué devuelve `ruta` si el origen y el destino son el mismo enrutador? ¿Es lo que el enunciado quiere?
2. En Poor Trade Advisor, ¿qué pasa si todas las carreteras tienen la misma PPA? ¿Y si hay una sola ciudad con carreteras de PPA máxima que la unen consigo misma?
3. Escriba *DFSAux* y *BFS* sobre matriz de adyacencia y sobre lista de aristas y diga cuánto cuesta cada recorrido en términos de V y E.

Lo que sigue

Lo que los recorridos todavía no dicen

La profundidad deja un árbol y clasifica las aristas que no usó: unas apuntan a un ancestro, otras a un descendiente, otras cruzan. Con esa clasificación se detectan ciclos, se ordena un grafo dirigido y se encuentran los puntos por donde un grafo se parte.

Para investigar

Busque qué es una **arista de retroceso** en un recorrido en profundidad, y qué tiene que ver con que un grafo dirigido tenga ciclos.

Contenido

1 Recuento

2 Grafos implícitos: Jaimico y el café

3 UVa 10977: el bosque encantado

4 UVa 627: la ruta con menos saltos

5 UVa 11749: el componente más grande

6 Errores comunes

7 Ejercicios

8 Referencias

Referencias

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms*, 3rd ed. MIT Press, 2009. Sección 22.2 (pp. 594–602) y Sección 22.3 (pp. 603–612).
- Halim, S., Halim, F., Effendy, S. *Competitive Programming 4*. Lulu, 2020. Sección 4.2, recorridos, y su tratamiento de las cuadrículas como grafos implícitos.
- Universidad de Valladolid Online Judge: problemas 627, 10977 y 11749. `https://vjudge.net`