Algoritmos con Python

El camino más corto

4 de marzo de 2021

Índice

Introducción

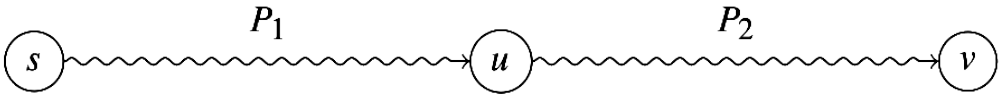
Un problema clásico de los grafos consiste en encontrar un camino más corto entre dos vértices, una fuente s y un destino v. Por el mismo costo podemos obtener los caminos más cortos entre la fuente s y todos los posibles destinos v'; es por eso que los algoritmos presentados en este capítulo resuelven este problema más general de los caminos más cortos desde una fuente única en un gráfico dirigido.

La longitud de una ruta se define como la suma de sus pesos de arco, y la distancia de s a v se define como la longitud de una ruta más corta entre s y v. Para simplificar la presentación, en general mostramos solo cómo calcular la distancias. Para obtener un camino que se dé cuenta de esta distancia, basta con mantener, además del conjunto de distancias, un conjunto de predecesores. Por lo tanto, si $\mathrm{dist}[v]$ es la distancia de s a v con $\mathrm{dist}[v] = \mathrm{dist}[u] + w[u][v]$ para un vértice u, almacenamos $\mathrm{prec}[v] = u$ en la matriz de predecesores . Si seguimos a los predecesores hasta el origen, podemos determinar la ruta más corta en orden inverso, desde el origen hasta un destino determinado.

1. Propiedad de la composición

Los caminos más cortos exhiben una propiedad de composición, que es la clave para los diferentes algoritmos del camino más corto. Richard Bellman llamó a esto el principio de la optimización, que se encuentra en el corazón de los problemas de la programación dinámica. Considere una ruta P de s a v (también conocida como ruta s - v), que pasa por un vértice u, consulte la figura 1. Por lo tanto, es la concatenación de una ruta P_1 de s a u con una ruta P_2 de u a v. La longitud de P es la suma de las longitudes de P_1 y P_2. Por lo tanto, si P es una ruta más corta de s a v, entonces P_1 debe ser una ruta más corta de s a u y P_2 una ruta más corta de u a v. Esto es obvio, ya que el reemplazo de P_1 por una ruta más corta daría como resultado un camino más corto de s a v.

Figura 1: Ruta más corta



Una ruta más corta de s a v que pasa por u es la concatenación de una ruta más corta de s a u con una ruta más corta de u a v.

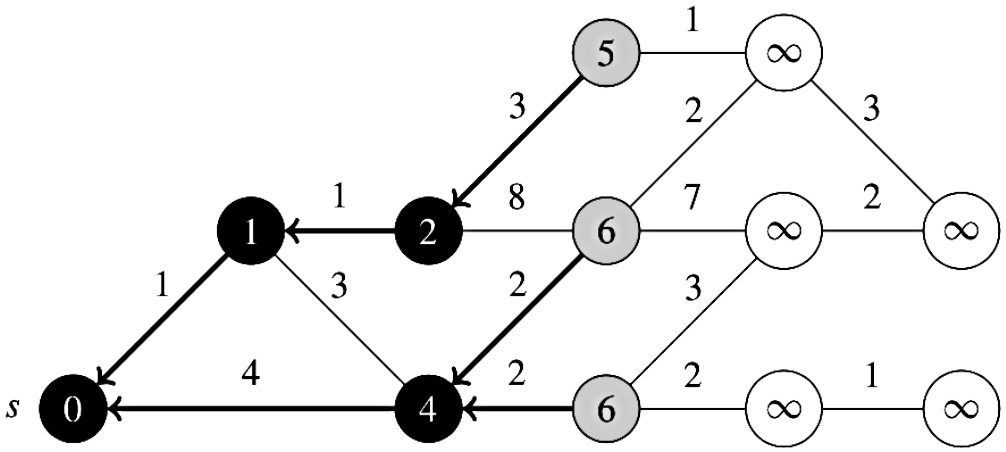
1.1. Vértices negros, grises y blancos

Esta propiedad de composición es la base del algoritmo de Dijkstra y sus variantes, incluso en grafos con pesos no negativos en sus arcos. El algoritmo mantiene una matriz `dist` que contiene las distancias desde la fuente hasta los vértices; se establece en $+\infty$ para los vértices v para los que aún no se ha encontrado ninguna ruta s - v.

Por tanto, los vértices del grafo se dividen en tres grupos, véase la figura 2. Los vértices negros son aquellos con una ruta más corta ya identificada desde la fuente, los vértices grises son los vecinos directos de los vértices negros y los vértices blancos son los que actualmente no tienen una ruta más corta conocida.

Inicialmente, solo la fuente s es negra, con $\mathrm{dist}[\mathrm{s}] = 0$. Todos los vecinos directos v de s son grises, y $\mathrm{dist}[v]$ es el peso del arco `(s, v)`. Todos los demás vértices son por el momento blancos. Luego, el algoritmo colorea repetidamente los vértices grises en negro y sus vecinos blancos en gris, manteniendo este invariante. Finalmente, todos los vértices alcanzables desde s serán negros, mientras que los demás permanecerán blancos.

Figura 2: Coloración de vértices



Ejemplo de coloración de vértices mediante el algoritmo de Dijkstra. Cada vértice v está etiquetado con $\mathrm{dist}[v]$, su distancia actual desde la fuente y los arcos especificados por `prec` están resaltados en negrita.

1.2. Observación clave

¿Qué vértice gris deberíamos seleccionar para colorear negro? La elección se hace con un vértice v con el menor $\mathrm{dist}[v]$. ¿Por qué es esta la elección correcta? Considere una ruta P arbitraria s-v. Dado que s es negro y v es gris, existe un primer vértice gris u en este camino. Por lo tanto, P puede descomponerse en una ruta $s - u$ P_1 y una ruta a $u - v$ P_2, donde P_1 contiene solo vértices negros intermedios, excluyendo u. Por la elección de v, $\mathrm{dist}[u] \geqslant \mathrm{dist}[v]$, y por la hipótesis de que los pesos de los arcos no son negativos, la longitud de P_2 tampoco es negativa. Por tanto, la longitud de P es al menos $\mathrm{dist}[v]$. Como la elección de P fue arbitraria, esto muestra que la ruta s - v más corta es de longitud $\mathrm{dist}[v]$. Por lo tanto, colorear v negro es válido, y para mantener el invariante, para cada vértice v' alcanzable desde v por un arco,

1.3. Estructura de datos para los vértices grises

Como para cada iteración buscamos un vértice gris v con $\mathrm{dist}[v]$ mínimo, una cola de prioridad que contenga los vértices v es adecuada, siendo los valores de prioridad los valores de `dist`. Esta es la implementación elegida para el algoritmo de Dijkstra. La eliminación del vértice con la menor distancia se puede hacer así en el tiempo logarítmico en el número de vértices.

Si el grafo tiene una estructura especial, entonces posiblemente se podría usar una estructura más simple. Por ejemplo, en el caso de que los pesos de los arcos estén en `{0,1}`, solo habrá dos tipos de vértices grises, los de distancia d y los de distancia $d + 1$. Esta cola de prioridad se puede implementar de una manera más sencilla utilizando una cola de dos extremos, abreviada deque. Esta cola contiene la lista de vértices grises en orden de prioridad, y basta con quitar de la izquierda en tiempo constante un vértice v para colorearlo de negro.

Los vecinos de v se agregan a la cola de la izquierda o de la derecha, según si el peso en el arco correspondiente es `0` o `1`. Por último, todas las operaciones de manipulación de la cola se pueden realizar en tiempo constante, y reducimos un factor logarítmico en comparación con el algoritmo de Dijkstra.

Si el grafo es aún más simple, con pesos unitarios en todos sus arcos, en una forma de decir un grafo no ponderado, entonces esta cola de dos extremos puede reemplazarse por una cola simple y administrarse mediante el algoritmo de amplitud primero.

Shortest Path Algorithms		
Without weights	$O(\|E\|)$	breadth-first search (BFS)
Weights 0 or 1	$O(\|E\|)$	Dijkstra adapted with a double-ended queue
Weights non-negative	$O(\|E\| \log \|V\|)$	Dijkstra
Weights arbitrary	$O(\|V\| \cdot \|E\|)$	Bellman-Ford
Paths from all sources	$O(\|V\|^3)$	Floyd-Warshall

2. Gráficos con pesos 0 o 1

2.1. Definición

Dado un gráfico cuyos arcos están ponderados por `0` o `1`, y un vértice fuente s, deseamos calcular las distancias desde s hasta los otros vértices.

2.2. Aplicación

Se le proporciona un mapa de un laberinto rectangular NxM que contiene paredes que obstruyen y le gustaría salir mientras demuele un número mínimo de paredes. El laberinto puede verse como un gráfico dirigido cuyos arcos (de una celda a una celda adyacente) están ponderados por `0` (acceso libre a la celda) o por `1` (acceso a una celda bloqueada por una pared). Buscamos derribar el menor número posible de muros, lo que se reduce a encontrar el camino más corto en este gráfico desde la entrada hasta la salida.

2.3. Algoritmo

Volvemos a la estructura de un algoritmo genérico de ruta más corta. En cualquier momento, los vértices del gráfico se dividen en tres grupos: negro, gris y blanco.

El algoritmo mantiene una cola de dos extremos que contiene todos los vértices grises, así como eventualmente algunos vértices que eran grises en el momento de su inserción, pero que desde entonces se han vuelto negros. La cola tiene la propiedad de que para un cierto valor x, todos los vértices negros v_{b} satisfacen $\mathrm{dist}\,[v_{\mathrm{b}}] = x$, y hasta cierta posición todos los vértices grises v_{g0} satisfacen $\mathrm{dist}\,[v_{\mathrm{g0}}] = x$, mientras que los siguientes satisfacen $\mathrm{dist}\,[v_{\mathrm{g1}}] = x + 1$.

Siempre que esta cola no esté vacía, el algoritmo extrae un vértice v del encabezado de la cola, que por lo tanto tiene un valor mínimo $\mathrm{dist}\,[v]$. Si este vértice ya es negro, no hay nada que hacer. De lo contrario, es de color negro. Para mantener el invariante, debemos agregar ciertos vecinos v' de v a la cola. Sea $\ell = \mathrm{dist}\,[v] + w\,[v]\,[v']$. Si v' ya es negro, o si $\mathrm{dist}\,[v'] \leqslant \ell$, no hay razón para agregar v' a la cola. De lo contrario, v' puede ser de color gris, $\mathrm{dist}\,[v']$ se reduce a ℓ y v' se agrega al principio o al final de la cola, dependiendo de si $w\,[v]\,[v'] = 0$ o $w\,[v]\,[v'] = 1$.

```
def dist01(graph, weight, source=0, target=None):
    n = len(graph)
    dist = [float(inf)] * n
    prec = [None] * n
```

```
    black = [False] * n
    dist[source] = 0
    gray = deque([source])
    while gray:
        node = gray.pop()
        if black[node]:
            continue
        black[node] = True
        if node == target:
            break
        for neighbor in graph[node]:
            ell = dist[node] + weight[node][neighbor]
            if black[neighbor] or dist[neighbor] <= ell:
                continue
            dist[neighbor] = ell
            prec[neighbor] = node
            if weight[node][neighbor] == 0:
                gray.append(neighbor)
            else:
                gray.appendleft(neighbor)
    return dist, prec
```

3. Grafos con ponderaciones no negativas — Dijkstra

3.1. Definición

Dado un grafo dirigido con pesos no negativos en sus arcos, buscamos el camino más corto entre una fuente y un destino.

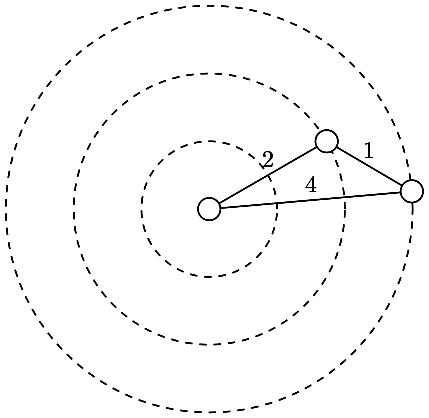
3.2. Complejidad

Hay una implementación brutal en $\mathrm{O}(|V|^2)$, mientras que el uso de una cola de prioridad reduce la complejidad a $\mathrm{O}(|\mathrm{E}|\log|V|)$. Usando una cola de prioridad de Fibonacci, podemos obtener una complejidad de $\mathrm{O}(|\mathrm{E}|+|V|\log|V|)$; sin embargo, la mejora tal vez no valga la pena el esfuerzo de implementación.

3.3. Algoritmo

El algoritmo de Dijkstra mantiene un conjunto de vértices S para los cuales ya se ha calculado una ruta más corta desde la fuente. Por tanto, S es precisamente el conjunto de vértices negros. Inicialmente, S contiene solo la fuente en sí. Además, el algoritmo mantiene un árbol de caminos más cortos enraizados en la fuente y cubriendo S. Asignamos $prec[v]$ como el predecesor de v en este árbol y $dist[v]$ como la distancia calculada, ver Figura 3.

Figura 3: Algoritmo de Dijkstra



El algoritmo de Dijkstra captura los vértices mediante bolas de radio creciente centradas en el vértice de salida. No hay vértices en la distancia 1, por lo que el algoritmo salta directamente al vértice más cercano, en la distancia 2, a través de la cola de prioridad.

Luego examinamos los bordes en el límite de S. Más precisamente, consideramos los arcos `(u, v)` con u en S y v no en S. Estos arcos definen caminos compuestos por un camino más corto desde la fuente hasta u seguido por el arco `(u, v)`. Este camino tiene un peso que le asignamos como prioridad al arco `(u, v)`.

El algoritmo extrae de la cola de prioridad un arco `(u, v)` de prioridad mínima, definiendo así una ruta `s - v` más corta. Luego podemos agregar el arco `(u, v)` al árbol de caminos más cortos, agregar v a S e iterar.

3.4. Colas de prioridad

El ingrediente principal de este algoritmo es una cola de prioridad, que es una estructura de datos que contiene un conjunto de elementos, lo que permite la adición de elementos y la extracción del que tiene la menor prioridad. Tal estructura se implementa, en general, con una pila, donde estas operaciones tienen un costo logarítmico en el tamaño del conjunto.

3.5. Una ligera mejora

Para mayor eficiencia, podemos ignorar los arcos que ya sabemos que no conducen a una ruta más corta y no almacenarlos en la cola de prioridad. Para esto, actualizamos $\mathrm{dist}[v]$ cada vez que se agrega un arco a v a la cola. Por tanto, al momento de considerar un arco, podemos decidir si mejora el camino más corto a v conocido hasta este punto.

3.6. Detalles de implementacion

El código presentado permite el cálculo de las rutas más cortas a todos y cada uno de los destinos, si no se especifica un destino en particular cuando se invoca la función. Para cada arco saliente `(u, v)`, almacenamos en la cola el par `(d, v)`, donde d es la longitud del camino asociado con este arco.

Es posible que un vértice se encuentre varias veces en la cola con diferentes pesos (prioridades). Sin embargo, una vez que se extrae la primera aparición de este vértice, se coloreará de negro y las demás apariciones se ignorarán en el momento de su extracción.

```
def dijkstra(graph, weight, source=0, target=None):
    n = len(graph)
    assert all(weight[u][v] >= 0 for u in range(n) for v in graph[u])
    prec = [None] * n
    black = [False] * n
    dist = [float(inf)] * n
    dist[source] = 0
    heap = [(0, source)]
    while heap:
```

```
        dist_node, node = heappop(heap)       # Closest node from source
        if not black[node]:
            black[node] = True
            if node == target:
                break
            for neighbor in graph[node]:
                dist_neighbor = dist_node + weight[node][neighbor]
                if dist_neighbor < dist[neighbor]:
                    dist[neighbor] = dist_neighbor
                    prec[neighbor] = node
                    heappush(heap, (dist_neighbor, neighbor))
    return dist, prec
```

3.7. Variante

La implementación del algoritmo de Dijkstra se puede simplificar ligeramente con el uso de una cola de prioridad que permite cambiar la prioridad de un elemento. En lugar de almacenar los arcos en la cola, almacena los vértices, de hecho todos los vértices del grafo, y la prioridad de un vértice v se establece en $\mathrm{dist}[v]$. En la práctica, la cola contiene pares de la forma $(\mathrm{dist}[v], v)$ y los comparamos lexicográficamente. Cuando se descubre una ruta más corta a v, actualizamos el par correspondiente a v con la nueva distancia más corta. La ventaja de esta variante es que podemos saltarnos marcando los vértices en negro. Para cada vértice v, la cola contiene solo un par con v, y cuando $(\mathrm{dist}[v], v)$ se extrae de la cola de prioridad, sabemos que hemos descubierto un camino más corto hacia v.

```
def dijkstra_update_heap(graph, weight, source=0, target=None):
    n = len(graph)
    assert all(weight[u][v] >= 0 for u in range(n) for v in graph[u])
    prec = [None] * n
    dist = [float(inf)] * n
    dist[source] = 0
    heap = OurHeap([(dist[node], node) for node in range(n)])
    while heap:
        dist_node, node = heap.pop()       # Closest node from source
        if node == target:
            break
        for neighbor in graph[node]:
            old = dist[neighbor]
            new = dist_node + weight[node][neighbor]
            if new < old:
```

```
            dist[neighbor] = new
            prec[neighbor] = node
            heap.update((old, neighbor), (new, neighbor))
    return dist, prec
```

4. Gráficas con pesos arbitrarios — Bellman – Ford

4.1. Definición

Para este problema, se permiten pesos de arco negativos. Si se puede alcanzar un ciclo de peso total negativo desde la fuente, la distancia desde la fuente a los vértices del ciclo es $-\infty$. De hecho, al atravesar este ciclo un número arbitrario de veces, obtenemos un camino desde la fuente hasta un vértice en el ciclo de peso negativo con longitud negativa y magnitud arbitrariamente grande. Esta molesta situación es detectada por el algoritmo presentado.

4.2. Complejidad

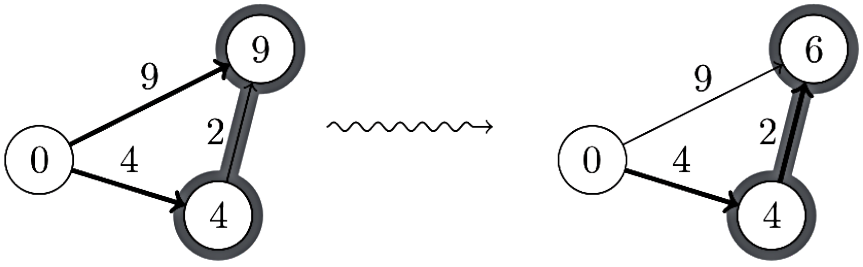
La programación dinámica proporciona una solución en $O(|V| \cdot |E|)$.

4.3. Algoritmo

La operación central es la relajación de las distancias (ver Figura 4), que consiste para cada arco `(u, v)` en probar si el uso de este arco podría disminuir la distancia de la fuente a v. El valor $d_u + w_{uv}$ es entonces un candidato para la distancia d_v. Esto se realiza con dos bucles anidados; el bucle interior relaja las distancias a través de cada arco y el bucle exterior repite este procedimiento un cierto número de veces. Demostramos que después de k iteraciones del bucle exterior, hemos calculado para cada vértice v la longitud de un camino más corto desde la fuente a v, utilizando como máximo k arcos. Esto es cierto para `k = 0`, y si asignamos $d_k[v]$ como esta longitud para cada $k = 1, \ldots, |V| - 1$, luego

$$d_{k+1}[v] = \min_{u:(u,v)\in E} d_k[u] + w_{uv}.$$

Figura 4: Relajación



Un ejemplo de relajación a través del arco gris. Seguir este borde acorta la ruta hasta el vértice de destino del arco. Los vértices están etiquetados con la distancia desde el vértice de origen, que tiene la etiqueta `0`. Las rutas más cortas se muestran en negrita.

4.4. Detección de ciclos negativos

Considere la situación en la que el gráfico no contiene ciclos negativos. En este caso, todos los caminos más cortos son simples y $|V| - 1$ iteraciones del bucle exterior son suficientes para determinar las distancias hacia cada uno de los destinos. Por lo tanto, si observamos un cambio durante la $|V|$-ésima iteración, esto indica que existe un ciclo negativo que es accesible desde la fuente. Nuestra implementación devuelve un booleano que indica la presencia de dicho ciclo.

```
def bellman_ford(graph, weight, source=0):
    n = len(graph)
    dist = [float(inf)] * n
    prec = [None] * n
    dist[source] = 0
    for nb_iterations in range(n):
        changed = False
        for node in range(n):
            for neighbor in graph[node]:
                alt = dist[node] + weight[node][neighbor]
                if alt < dist[neighbor]:
                    dist[neighbor] = alt
                    prec[neighbor] = node
                    changed = True
        if not changed:                    # fixed point
            return dist, prec, False
    return dist, prec, True
```

5. Todas las rutas de origen-destino: Floyd-Warshall

5.1. Definición

Dado un gráfico con pesos en sus arcos, queremos calcular la ruta más corta entre cada par de nodos. Una vez más, el problema está bien planteado solo en ausencia de ciclos con peso negativo, y el algoritmo detecta esta excepción.

5.2. Complejidad

El algoritmo de Floyd-Warshall se ejecuta en $O(n^3)$, donde n es el número de vértices.

5.3. Algoritmo

Las distancias entre los vértices se calculan mediante programación dinámica, consulte la Figura 5. Para cada $k = 0, 1, \ldots, n$ se calcula una matriz cuadrada W_k, que contiene en $W_k[u][v]$ la longitud de la ruta más corta de u a v utilizando solo los vértices intermedios del índice estrictamente inferior a k, donde los vértices se numeran de `0` a `n - 1`. Por tanto, para $k = 0$, la matriz W_0 contiene solo los pesos de los arcos, y $+\infty$ para los elementos (u, v) que no tienen arco (u, v). La actualización es simple usando el principio de composición: una ruta más corta de u a v que pasa por un vértice k se compone de una ruta más corta de u a k y una ruta más corta de k a v. Por lo tanto, calculamos, para $k, u, v \in \{0, \ldots, n-1\}$:

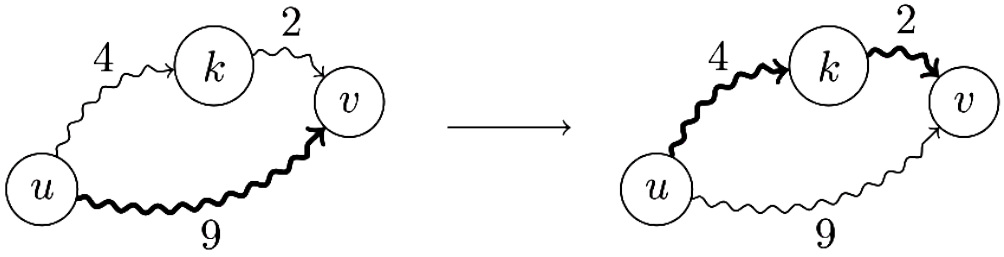
$$W_{k+1}[u][v] = \min\{W[u][v],\ W_k[u][k] + W_k[k][v]\}.$$

De hecho, es el mismo k como índice de los vértices y como posición en la matriz: para el cálculo de $W_{k+1}[u][v]$, podemos considerar los caminos que pasan por el vértice k.

$$a_{uv} = \sum_{k=0}^{n-1} b_{uk} \times c_{kv} \qquad W_{uk}^{(\ell+1)} = \min_{k=0}^{n-1} W_{uk}^{\ell} + W_{kv}^{\ell} = 0$$

$$A = BC \qquad W^{(\ell+1)} = W^{(\ell)} W^{(\ell)}$$

Figura 5: Un ejemplo de relajación a través del vértice k



Pasar por k acorta la ruta de u a v

5.4. Detalles de implementacion

La implementación mantiene una matriz `W` única, que desempeña el papel de la W_k sucesiva; modifica la matriz de peso dada. Detecta la presencia de ciclos negativos y devuelve `False` en este caso.

```
def floyd_warshall(weight):
    V = range(len(weight))
    for k in V:
        for u in V:
            for v in V:
                weight[u][v] = min(weight[u][v],
                                   weight[u][k] + weight[k][v])
    for v in V:
        if weight[v][v] < 0:      # negative cycle found
            return True
    return False
```

5.5. Detección de ciclos negativos

Existe un ciclo negativo que pasa por un vértice v si y solo si $W_n[v][v] < 0$. Sin embargo, se recomienda encarecidamente detectar los ciclos negativos con el algoritmo Bellman-Ford. De hecho, en presencia de ciclos negativos, los valores calculados por el algoritmo Floyd-Warshall pueden tener un valor absoluto exponencialmente grande en el número de vértices del gráfico, provocando un desbordamiento en la capacidad de las variables.

5.6. Ejemplo: Rutas en Berland

Suponga que se nos da un grafo dirigido fuertemente conectado con pesos no negativos en los arcos. Éstos definen una distancia más corta $D[u, v]$ para cada par de vértices `(u, v)`, que puede calcularse mediante el algoritmo Floyd-Warshall. La puntuación del gráfico se define como la suma de las distancias más cortas en todos los pares de vértices. Ahora, suponga que se agrega un arco adicional (u', v') con peso $w[u', v']$ al grafo. Por ejemplo, se construye un túnel a través de una montaña desde u' hasta v', mucho más corto que el antiguo camino que rodea la montaña entre u' y v'. ¿En cuánto se reducirá la puntuación total en presencia de este nuevo arco? Para ello, basta con calcular la suma $\sum_{u,v} \max\{0, D[u, v] - (D[u, u'] + w[u', v'] + D[v', v])\}$. Si n es el número de vértices del gráfico y k el número de arcos adicionales, entonces la complejidad es $O(n^3 + kn^2)$, bastante aceptable para los límites $n, k \leqslant 300$.

6. Cuadrícula

6.1. Problema

Dada una cuadrícula rectangular, donde ciertas celdas son inaccesibles, encuentre un camino más corto entre la entrada y la salida.

6.2. El algoritmo

Este problema se resuelve fácilmente con una búsqueda de amplitud en el gráfico determinado por la cuadrícula. Sin embargo, en lugar de construir explícitamente el gráfico, es más sencillo realizar la búsqueda directamente en la cuadrícula. La cuadrícula se presenta en forma de matriz bidimensional que contiene el carácter # para las celdas inaccesibles y un espacio para las celdas accesibles.

Nuestra implementación usa esta misma matriz para marcar los vértices ya visitados y, por lo tanto, evita una estructura de datos adicional. Las celdas visitadas contendrán los símbolos '→', '←', '↓', '↑', que indican el predecesor en la ruta más corta desde la fuente.

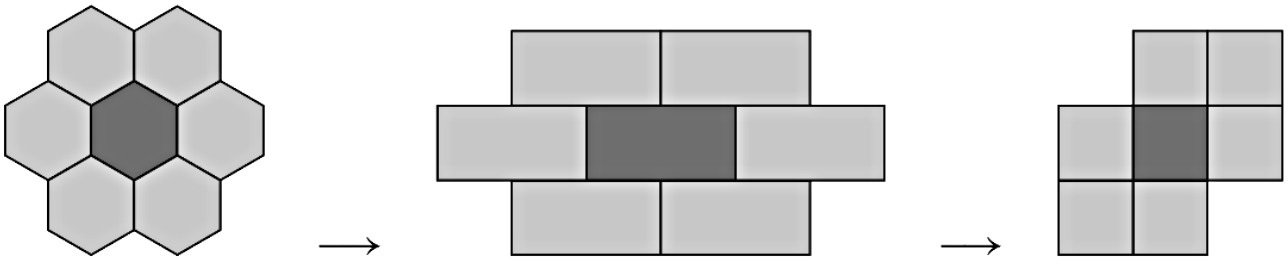
```
def dist_grid(grid, source, target=None):
    rows = len(grid)
    cols = len(grid[0])
    i, j = source
    grid[i][j] = s
    Q = deque()
    Q.append(source)
    while Q:
        i1, j1 = Q.popleft()
        for di, dj, symbol in [(0, +1, >),
                               (0, -1, <),
                               (+1, 0, v),
                               (-1, 0, ^)]:   # explore all directions
            i2 = i1 + di
            j2 = j1 + dj
            if not (0 <= i2 < rows and 0 <= j2 < cols):
                continue              # reached the bounds of the grid
            if grid[i2][j2] !=  :   # inaccessible or already visited
                continue
            grid[i2][j2] = symbol     # mark visit
            if (i2, j2) == target:
                grid[i2][j2] = t    # goal is reached
```

```
                return
            Q.append((i2, j2))
```

6.3. Variantes

La implementación propuesta es fácil de modificar para permitir pasos diagonales en la cuadrícula, entre celdas que comparten solo una esquina. Una cuadrícula hexagonal se puede tratar de manera similar transformándola en una cuadrícula con celdas cuadradas, pero con una vecindad particular. La figura 6 ilustra esta transformación.

Figura 6: Cuadrícula hexagonal



Representación de una cuadrícula hexagonal mediante una cuadrícula cuadrada.

7. Variantes

Dada la importancia del problema del camino más corto, presentamos una serie de variantes clásicas.

7.1. Grafos no ponderados

En un grafo no ponderado, una búsqueda en amplitud es suficiente para determinar las rutas más cortas.

7.2. Grafos acíclicos dirigidos

Una ordenación topológica, permite procesar los vértices en un orden apropiado: para calcular la distancia desde la fuente a un vértice v, podemos beneficiarnos de las distancias a todos los predecesores de v calculadas de antemano y aplicar una recursividad de programación dinámica simple.

Aplicación

Deseamos determinar una ruta de casa al trabajo que comienza en ascenso y termina en descenso, para primero hacer algo de ejercicio y luego refrescarse. Para ello, modelamos la ciudad como un grafo, donde los vértices representan las intersecciones con su elevación y los bordes representan las carreteras con sus longitudes. Usando el algoritmo anterior, podemos calcular para cada vértice v la distancia desde la fuente usando solo arcos ascendentes, y al mismo tiempo calcular la distancia desde v hasta el objetivo usando solo arcos descendentes. La respuesta es la suma de estas distancias minimizadas en todos los vértices v.

7.3. Camino más largo

La programación dinámica mencionada anteriormente se puede adaptar para calcular de manera eficiente las rutas más largas en un grafo acíclico dirigido (DAG para abreviar). Para un grafo general, el problema consiste en encontrar el camino más largo desde el origen al destino que pasa por cada vértice como máximo una vez. Este problema es NP-Hard; no conocemos un algoritmo para resolverlo en tiempo polinomial. Si el número de vértices es pequeño, del orden de veinte más o menos, entonces podemos usar el método de programación dinámica en los subconjuntos S de vértices y calcular $\mathrm{D}[S][v]$, la longitud del camino más largo desde la fuente a v usando solo los vértices intermedios de S. Entonces, para cada S no vacío, tenemos la relación

$$D[S][v] = \max_{u \in S} D[S \setminus u][u] + w[u][v],$$

donde $S \setminus u$ es el conjunto S con el vértice u eliminado, y $w[u][v]$ es el peso del arco (u, v). El caso base es entonces

$$D[\emptyset][v] = \begin{cases} w[u][v] & \text{si el arco (u,v) existe} \\ -\infty & \text{en caso contrario.} \end{cases}$$

7.4. El camino más largo de un árbol

A menudo, los problemas que son difíciles en general se vuelven fáciles para los árboles, ya que la noción de subárbol puede permitir una solución mediante programación dinámica. Este es de hecho el caso del problema de la ruta más larga en un árbol.

7.5. Trayectoria que minimiza el peso máximo sobre los arcos

En el caso de que el objetivo no sea minimizar la suma de los pesos de los bordes de una ruta, sino el máximo, una solución simple es basarse en el algoritmo de Kruskal, que utiliza una estructura de búsqueda de unión. Comenzamos con un grafo vacío y agregamos los bordes en orden de peso creciente, hasta que el origen y el destino se unen en el mismo componente conectado. Para el grafo dirigido, se puede hacer que funcione un enfoque similar, pero la solución más simple podría ser adaptar el algoritmo de Dijkstra, reemplazando las adiciones de distancias con el máximo de pesos de arco en una ruta.

7.6. Grafo ponderado en los vértices

Considere un grafo cuyos vértices tienen pesos, en lugar de arcos. El objetivo es encontrar un camino desde el origen hasta el destino minimizando la suma de los pesos de los vértices atravesados. Esta situación se puede reducir al caso descrito anteriormente. Reemplazamos cada vértice v por dos vértices v^- y v^+ unidos por un arco con el peso de v, y reemplazamos cada arco (u, v) por un arco (u^+, v^-) de peso cero.

7.7. Ruta que minimiza el peso máximo en los vértices

En el caso de que el peso de un camino se defina como el máximo de los pesos de los vértices intermedios del camino, podemos utilizar el algoritmo Floyd-Warshall. Basta ordenar

los vértices aumentando el peso y terminar la iteración tan pronto como se encuentre una ruta desde el origen al destino.

Basta con mantener una matriz booleana de conectividad $C_k[u][v]$ que indica si existe una ruta de u a v que utilice sólo los vértices intermedios con índices estrictamente inferiores a k. La actualización se convierte así

$$C_k[u][v] = C_{k-1}[u][v] \vee (C_{k-1}[u][k] \wedge C_{k-1}[k][v]).$$

7.8. Todos los bordes que pertenecen a un camino más corto

Dado un grafo ponderado `G`, una fuente `s` y un destino `t`, pueden existir varias rutas más cortas entre `s` y `t`. El objetivo es determinar todos los bordes que pertenecen a estos caminos más cortos. Para lograr esto, hacemos dos invocaciones del algoritmo de Dijkstra, una con `s` y la otra con `t` como fuente, para calcular para cada vértice `v` la distancia $d[s, v]$ desde la fuente y la distancia $d[v, t]$ hacia el destino. Una arista `(u, v)` de peso $w[u, v]$ pertenecerá a un camino más corto si y solo si

$$d[s, u] + w[u, v] + d[v, t] = d[s, t]$$

Variante para un grafo dirigido

El algoritmo de Dijkstra admite solo una fuente determinada y no un destino específico. Por tanto, para el cálculo de todas las distancias $d[v, t]$ hacia `t`, es necesario invertir temporalmente los arcos.