

Tema-3.-Algoritmos-voraces.pdf



josee_mjr



Diseño y Análisis de Algoritmos



3º Grado en Ingeniería de la Ciberseguridad



Escuela Técnica Superior de Ingeniería Informática. Campus de
Móstoles
Universidad Rey Juan Carlos



Estamos de
Aniversario

De la universidad al
mercado laboral:
especialízate con los posgrados
de EOI y marca la diferencia.



EOI Escuela de
organización
industrial



saber más



Organiza la mejor fiesta universitaria

Organiza cualquier fiesta en CircusAlcala. Solo contáctanos y crearemos el evento perfecto para ti y tus amigos.



Escanea y únete a la fiesta en CircusAlcala. ¡Diversión asegurada en el mejor club!

Tema 3. Algoritmos voraces

- [Motivación: Problema de la devolución del cambio](#)
- [Esquema de la técnica](#)
- [Aspectos de diseño](#)
- [Aplicaciones de los algoritmos voraces](#)
 - [Minimización del tiempo en el sistema](#)
 - [Problema de la Mochila](#)
 - [Planificación con plazo fijo](#)
 - [Problemas en grafos \(árboles de recubrimiento y caminos más cortos\)](#)
 - [Árboles de recubrimiento \(expansión/generador\) mínimo](#)
 - [Algoritmo de Kruskal](#)
 - [Algoritmo de Prim](#)
 - [Caminos mínimos](#)
 - [Algoritmo de Dijkstra](#)
 - [Heurísticas voraces](#)
 - [Coloreado de grafos](#)
 - [Viajante de comercio](#)

Motivación: Problema de la devolución del cambio

Dado un problema con n entradas, el objetivo es obtener un subconjunto de estas, de tal forma que se satisfaga una determinada restricción de forma óptima. La forma habitual de resolverlo es escoger las mejores entradas que verifiquen las restricciones hasta que se encuentra la solución que se busca.

☰ Problema del cambio

Supongamos un país cuyo sistema monetario tiene monedas con valores v_1, v_2, \dots, v_n . El problema del cambio se pueden enunciar como:

"Descomponer una cantidad dada M , en monedas de valores v_1, v_2, \dots, v_n , de forma que el número de monedas utilizados sea mínimo"

Datos relevantes:

- Candidatos: Monedas $C = v_1, v_2, \dots, v_n$
- Solución: La suma de las monedas elegidas son igual al cambio
- Factibilidad: la suma de monedas no puede superar al cambio
- Objetivo: minimizar las monedas devueltas
- ¿Selección? Moneda de mayor valor mientras sea factible

```
# Ejemplo de algoritmo-solución al problema del cambio
# Greedy algorithm for minimizing coins
def isFeasible(cambio, billete):
    return cambio // billete

def isSol(cambio):
    return cambio > 0

def greedyCoins(cand, cambio):
    sol = [0] * len(cand) # Se guarda la cantidad de billetes de cada tipo
    i = 0 # Indice para iterar sobre cada candidato
    while isSol(cambio) and i < len(cand): # Mientras cambio sea mayor que cero y no se acaben los
        candidatos
```



WUOLAH

```

        if not isFeasible(cambio, cand[i]):
            i += 1 # Si el cambio es indivisible por el candidato actual paso al siguiente
        else:
            sol[i] += 1 # Si es divisible sumo 1 a ese tipo de billete
            cambio -= cand[i] # Y le resto al cambio el valor de ese candidato

    return sol

# Main
cand = [500, 200, 100, 50, 20, 10, 5] # Sistema monetario ordenado de Mayor a Menor
cambio = 437
sol = greedyCoins(cand, cambio)
print(sol)

```

Esquema de la técnica

Identificar:

- **Conjunto de candidatos** y conjuntos de seleccionados
- **Función de selección**: elige el candidato idóneo en cada etapa
- **Función solución**: determina si los candidatos seleccionados son una solución
- **Función de factibilidad**: determina si el conjunto de seleccionados es prometedor
- **Función objetivo**: determina el valor de la solución

Características del esquema:

- Se construye una solución iterativamente
- Se toma la decisión óptima en cada iteración
- Una vez analizado un candidato (introducir o excluir), no se reconsidera la decisión
- Son voraces porque en cada etapa toman la mejor decisión sin preocuparse de mañana

Aspectos de diseño

Ventajas	Desventajas
La implementación de este tipo de algoritmos suele ser sencilla	No siempre se encuentra la solución óptima (por ejemplo, cambio con monedas de 11, 5 y 1)
Producen soluciones de forma muy eficiente (complejidad polinómica)	No reconsiderar decisiones pasadas puede conducir a no obtener el óptimo global (por ejemplo, el problema del viajante)
Encuentran la solución óptima para un número determinado de problemas	Encontrar la función de selección que garantice la optimalidad (por ejemplo, prob. de la mochila)
	Demostración formal de la optimalidad (encuentra el óptimo global)

Aplicaciones de los algoritmos voraces

Minimización del tiempo en el sistema

☰ Minimización del tiempo de espera

Supongamos un servidor que tiene que dar servicio a n clientes (procesador, cajero, ...). El tiempo requerido por cada cliente t_i es conocido. Se desea minimizar el tiempo medio de cada cliente en el sistema.

$$T_{med} = \frac{\sum_{i=0}^n t_i}{n}$$

Como n es conocido, equivale a minimizar el tiempo total invertido por cada cliente en el sistema

$$T = \sum_{i=0}^n t_i$$

Datos relevantes:

- Conjunto de candidatos: los n clientes
- Función solución: todos los clientes han sido ordenados
- Función de factibilidad: si han sido ordenados los clientes o no
- Función objetivo: minimizar T
- Función de selección: Elegir los candidatos por orden creciente de t_i . El algoritmo voraz se reduce a ordenar de forma no decreciente en t_i los n clientes.

```
import random
import sys

def get_best_task(tasks, candidates):
    best_task_time = sys.maxsize # Tambien valdria float('inf')
    best_task = None
    for c in candidates: # Por cada indice
        time = tasks[c] # Cuanto tarda la tarea con ese indice
        if time < best_task_time: # Selecciona la menor
            best_task_time = time
            best_task = c
    return best_task # Devuelve el indice de la menor

def greedy_waiting_time(tasks):
    result = []
    candidates = set()
    n = len(tasks)
    for i in range(n):
        candidates.add(i) # Los candidatos son todos los Indices de las tareas
    # Greedy Loop
    while candidates:
        best_task = get_best_task(tasks, candidates) # Se podría hacer solo con candidates si
        # fuera una copia de tasks
        candidates.remove(best_task) # Se van eliminando las mejores de los candidatos
        result.append(best_task)
    return result # Devuelve los indices de las mejores tareas ordenadas de menor a mayor segun el
    tiempo

# Main program
n = 10
tasks = []
# Generating a random array
for i in range(n):
    tasks.append(random.uniform(44, 140))
print(tasks)

# Greedy Waiting Time
print(greedy_waiting_time(tasks))
```

Problema de la Mochila

☰ Problema de la mochila

Supongamos que tenemos n objetos y una mochila. Cada objeto i tiene un peso $w_i > 0$ y un valor $v_i > 0$. La mochila puede llevar un peso que no sobrepase W . Se desea llenar la mochila maximizando el valor de los objetos transportados

$$\max \sum_{i=1}^n x_i v_i$$



Organiza la mejor fiesta universitaria

Organiza cualquier fiesta en CircusAlcala. Solo contáctanos y crearemos el evento perfecto para ti y tus amigos.



con las restricciones

$$\sum_{i=1}^n x_i w_i \leq W, 0 \leq x_i \leq 1 \text{ con } 1 \leq i \leq n$$

Datos relevantes:

- Conjunto de candidatos: los n objetos
- Función solución: cuando no puedan añadirse más fracciones de objetos a la mochila
- Función de factibilidad: $\sum_{i=1}^n x_i w_i \leq W$
- Función objetivo: maximizar $\max \sum_{i=1}^n x_i v_i$
- Función de selección: ¿?

```
def isFeasible(freeWeight, data, best_item):
    return (freeWeight - data['weight'][best_item] >= 0)

def get_best_item(data, candidates):
    best_ratio = -1
    for c in candidates:
        ratio = data['profit'][c] / data['weight'][c]
        if ratio > best_ratio:
            best_ratio = ratio
            best_item = c
    return best_item

def greedy_knapsack(data):
    n = len(data['profit'])
    sol = [0] * n
    candidates = set()
    for i in range(n):
        candidates.add(i)
    freeWeight = data['maxWeight']
    isSol = False
    # Si ordeno el array en este punto con quicksort el best_item sería O(1)
    # La complejidad del algoritmo voraz sería entonces O(n log n)
    while candidates and not isSol:
        best_item = get_best_item(data, candidates)
        candidates.remove(best_item)
        if isFeasible(freeWeight, data, best_item):
            sol[best_item] = 1
            freeWeight -= data['weight'][best_item]
        else:
            ratio = freeWeight / data['weight'][best_item]
            sol[best_item] = ratio
            isSol = True
    return sol

# Main program
data = {'profit': [20, 30, 66, 40, 60], 'weight': [10, 20, 30, 40, 50], 'maxWeight': 100}
print(greedy_knapsack(data))
```

Para un **problema de optimización** que se pueda resolver con una solución iterativa, un algoritmo voraz siempre va a encontrar una solución bastante efectiva, pero no siempre la optima. En el problema de las monedas no encuentra la optima, en la de los tiempos mínimos si encuentra el óptimo y en el de la mochila también encuentra la óptima siempre y cuando se usen ratios y se pueda partir el ultimo item en meter.

Planificación con plazo fijo

☰ Planificación con plazo fijo

WUOLAH



Supongamos que tenemos n trabajos, donde cada trabajo i tiene una fecha tope de realización $f_i > 0$ y un beneficio $b_i > 0$

- Para cualquier trabajo i , el beneficio b_i se gana si y sólo si se realiza antes (o coincidiendo) con su fecha tope f_i
- El trabajo se realiza en una máquina que consume una unidad de tiempo y sólo hay una máquina disponible (i.e., en un instante de tiempo sólo se puede ejecutar una tarea)

Datos relevantes:

- Conjunto de candidatos: los n trabajos a realizar
- Función solución: cuando se haya planificado todas las tareas
- Función de factibilidad: conjunto T de trabajos que todavía se pueden completar antes de su tope
- Función objetivo: maximizar $\sum_{i \in T} b_i$
- Función de selección: Considerar trabajos en orden decreciente de los beneficios

```
def get_best_item(data, candidates):
    best_item = -1
    best_profit = -1
    for c in candidates: # Por cada indice
        profit = data['profit'][c] # Miro su beneficio
        if profit > best_profit:
            best_profit = profit # Guardo el mayor beneficio
            best_item = c # Guardo su indice
    return best_item # Devuelvo el indice del trabajo con mayor beneficio dentro de los candidatos

def greedySchedule(data):
    n = len(data['profit']) # Numero de trabajos
    sol = [-1] * n
    candidates = set()
    for i in range(n):
        candidates.add(i) # Los candidatos son los indices de los trabajos

    last_date = max(data['deadline']) # La fecha limite será la mayor de todas a las deadlines
    j = 0
    while candidates and j <= last_date: # Mientras haya candidatos y todavía estemos en fecha
        best_item = get_best_item(data, candidates)
        candidates.remove(best_item) # Elimino de candidatos el indice del mejor en esa iteración
        i = data['deadline'][best_item] # Busco la deadline del trabajo con mejor profit en esa
iteracion
        found = False
        while i >= 0 and not found:
            if sol[i] == -1: # Busco un hueco libre donde meter el indice del mejor trabajo
en la solucion
                sol[i] = best_item
                found = True # Para salir del bucle si se ha encontrado hueco
            i -= 1 # Si no se ha encontrado se pasará al siguiente a la izquierda
        j += 1
    return sol

# Main program
data = {'profit': [50, 10, 15, 30],
        'deadline': [2, 1, 2, 1]}
print(greedySchedule(data))
```

Problemas en grafos (árboles de recubrimiento y caminos más cortos)

Árboles de recubrimiento (expansión/generador) mínimo

🔗 Problema



ÚNETE A McDONALD'S
Y ENCUENTRA A TU GENTE

¿TE VIENES?




My CREW
Mi trabajo. Mi pasión. Mi gente.

Dado un grafo $G = (V, E)$ no dirigido y ponderado con pesos positivos, calcular el subgrafo conexo $T \subseteq G$, que conecte todos los vértices del grafo G y que la suma de aristas seleccionadas sea mínima. Si el grafo es conexo, el subgrafo resultante es necesariamente un árbol

Hay dos estrategias para resolver este problema:

- Seleccionar la arista más corta en cada iteración → Algoritmo de Kruskal
- Seleccionar un vértice al azar y construir el árbol a partir de él, añadiendo las aristas de menor peso que tenga un extremo en la solución y otro no → Algoritmo de Prim

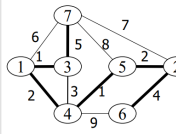
Algunos datos relevantes:

- **Candidatos:** conjunto de aristas E
- **Solución:** se ha construido un árbol con $|V|$ vértices y $|V| - 1$ aristas (extensible a grafos no conexos)
- **Factibilidad:** no existe ningún ciclo
- **Objetivo:** minimizar la suma de los pesos de las aristas seleccionadas.
- ¿Selección? Depende del algoritmo

Algoritmo de Kruskal

Primero ordena las aristas de mayor a menor peso. El árbol se construye a partir de varias componentes conexas. Sólo se incluye una arista si une dos componentes conexas. El algoritmo termina cuando sólo hay una componente conexa y este **encuentra la solución óptima al problema**. Lo que hace es comenzar ordenando las aristas de menor a mayor de acuerdo a su peso. Luego, va agregando aristas en orden, procurando que no se hagan ciclos. El algoritmo termina una vez que se hayan agregado $n - 1$ aristas. A diferencia del algoritmo de Prim, en el algoritmo de Kruskal no tiene la garantía de en cada paso tener una construcción parcial que sea conexa. Otra diferencia importante con Prim es que no se parte de ningún nodo específico. Aquí un código que implementa el algoritmo de una de las varias maneras que se puede implementar:

Ejemplo algoritmo de Kruskal



Paso	Arista	Comp.conexas
Inicial.	-----	$\{1\}\{2\}\{3\}\{4\}\{5\}\{6\}\{7\}$
1	$\{1,3\}$	$\{1,3\}\{2\}\{4\}\{5\}\{6\}\{7\}$
2	$\{4,5\}$	$\{1,3\}\{4,5\}\{2\}\{6\}\{7\}$
3	$\{1,4\}$	$\{1,3,4,5\}\{2\}\{6\}\{7\}$
4	$\{2,5\}$	$\{1,3,2,4,5\}\{6\}\{7\}$
5	$\{3,4\}$	Rechazada
6	$\{2,6\}$	$\{1,3,2,4,5,6\}\{7\}$
7	$\{3,7\}$	$\{1,3,2,4,5,6,7\}$

Conjunto de candidatos ordenado:
 $\{1,3\}, \{4,5\}, \{1,4\}, \{2,5\}, \{2,4\}, \{2,6\}, \{3,7\}, \{1,7\}, \{2,7\}, \{5,7\}, \{4,6\}$

```
def sort_candidates(g):
    candidates = []
    for node in g:
        for adj in g[node]:
            candidates.append(adj) # Se añade a la lista de candidatos cada una de las aristas del grafo
    candidates.sort(key=lambda tupla: tupla[2]) # Se ordenan los candidatos según el segundo item de la tupla,
    el peso
    return candidates

def update_components(components, newID, oldID): # Los nodos que tenían la etiqueta oldID pasan a tener newID
    for i in range(len(components)): # Con esta función vamos diferenciando las componentes
        if components[i] == oldID:
            components[i] = newID

def greedy_kruskal(g):
    candidates = sort_candidates(g) # Aristas ordenadas de menor a mayor peso
    sol = 0 # Suma de pesos mínima
    components = list(range(1, len(g.keys()) + 1)) # Lista con las etiquetas de las componentes
    n_comp = len(components) # Número de componentes conexas
    i = 0
    while i < len(candidates) and n_comp > 1: # Se itera mientras tengamos candidatos y el número de
    componentes > 1
        (start, end, weight) = candidates[i] # Seleccionamos un candidato
        if components[start-1] != components[end-1]: # Comprobamos que no es un bucle
            sol += weight
            n_comp -= 1
        i += 1
```




Organiza la mejor fiesta universitaria



Organiza cualquier fiesta en CircusAlcala. Solo contáctanos y crearemos el evento perfecto para ti y tus amigos.

```

update_components(components, components[start-1], components[end-1]) # Actualizamos las
componentes conexas
    i += 1
    return sol

# Main Program
g = {
    1: [(1, 3, 1), (1, 4, 2), (1, 7, 6)],
    2: [(2, 5, 2), (2, 6, 4), (2, 7, 7)],
    3: [(3, 1, 1), (3, 4, 3), (3, 7, 5)],
    4: [(4, 1, 2), (4, 3, 3), (4, 5, 1), (4, 6, 9)],
    5: [(5, 2, 2), (5, 4, 1), (5, 7, 8)],
    6: [(6, 2, 4), (6, 4, 9)],
    7: [(7, 1, 6), (7, 2, 7), (7, 3, 5), (7, 5, 8)]
}
print(greedy_kruskal(g))

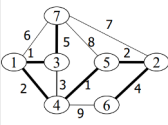
```

La complejidad de este algoritmo es $O(|V||E|)$. Además ocurre una fusión de componentes conexas con la estructura de datos *union-find*, es decir, si se copia el menor en el mayor, el análisis amortizado nos dice que la fusión se puede hacer en $O(\log |V|)$

Algoritmo de Prim

El árbol se construye a partir de una raíz. En cada iteración se añade una nueva rama (arista) al árbol que sólo se incluye una arista si no genera ciclos. A grandes rasgos, lo que hace es paso a paso elegir la mejor arista (la de peso mínimo) que una a un vértice del árbol en construcción, con uno que aún no esté en el árbol. Ya elegida, añade esa y el vértice al árbol en construcción. Se puede iniciar el algoritmo en cualquier vértice. El algoritmo termina cuando sólo se han añadido todos los vértices y encuentra la solución óptima al problema. Aquí un código que implementa el algoritmo de una de las varias maneras que se puede implementar:

Ejemplo algoritmo de Prim



Paso	Arista	Nodos que pertenecen al grafo
Inicial.	-----	{1}
1	{1,3}	{1,3}
2	{1,4}	{1,3,4}
3	{4,5}	{1,3,4,5}
4	{2,5}	{1,3,4,5}
5	{3,4}	Rechazada
6	{2,6}	{1,3,2,4,5,6}
7	{3,7}	{1,3,2,4,5,6,7}

Conjunto de aristas ordenado:
~~{1,3}~~, ~~{4,5}~~, ~~{1,4}~~, ~~{2,5}~~,
~~{3,4}~~, ~~{2,6}~~, ~~{3,7}~~, ~~{1,7}~~,
~~{2,7}~~, ~~{5,7}~~, ~~{4,6}~~

```

import random

def selectBestItem(cand, visit):
    minDist = float('inf')
    bestItem = None
    for i in range(len(cand)): # Por cada candidato
        if not visit[cand[i][1]-1] and cand[i][2] < minDist: # Compruebo si no ha sido visitado y busco la
            # arista con menor peso
            minDist = cand[i][2] # Menor peso
            bestItem = cand[i][1] # Índice del destino
    return minDist, bestItem

def greedy_prim(graph):
    size = len(graph)
    rand = random.randint(1, size) # Origen random
    candidates = graph[rand] # Inserto en la lista de candidatos los adyacentes al origen
    visitados = [False] * size # Suele ser más eficiente un array de bool
    visitados[rand - 1] = True # Origen visitado
    path = [rand]
    sol = 0
    for _ in range(1, size):
        cost, dest = selectBestItem(candidates, visitados) # Selecciono el mejor Item
        if dest is not None: # Compruebo que dest tenga un valor
            visitados[dest-1] = True # Vistio el destino
            path.append(dest) # Lo añado al path
            sol += cost
        for adj in graph[dest]:
            if not visitados[adj[1] - 1]:

```



Escanea y únete a la fiesta en CircusAlcala. ¡Diversión asegurada en el mejor club!

WUOLAH

```

        candidates.append(adj) # Por cada adyacente del destino añado a los candidatos
        # su arista si el nodo destino de esta no ha sido visitado

    print(path)
    return sol

g = {
    1: [(1, 3, 1), (1, 4, 2), (1, 7, 6)],
    2: [(2, 5, 2), (2, 6, 4), (2, 7, 7)],
    3: [(3, 1, 1), (3, 4, 3), (3, 7, 5)],
    4: [(4, 1, 2), (4, 3, 3), (4, 5, 1), (4, 6, 9)],
    5: [(5, 2, 2), (5, 4, 1), (5, 7, 8)],
    6: [(6, 2, 4), (6, 4, 9)],
    7: [(7, 1, 6), (7, 2, 7), (7, 3, 5), (7, 5, 8)]
}

print(greedy_prim(g))

```

El algoritmo tiene una complejidad de $O(|V||E|)$. Si se utiliza una cola de prioridad ordenada por el coste de las aristas, habrá un extremo en la solución parcial y otro fuera. La implementación eficiente garantiza $O(|A| \log |V|)$

Caminos mínimos

❓ Problema

Dado un grafo conexo $G = (V, E)$, dirigido y ponderado con pesos positivos, se toma uno, v , de los vértices como origen. El problema consiste en determinar la longitud mínima del camino que empieza en v hasta el resto

Hay dos estrategias para resolver este problema:

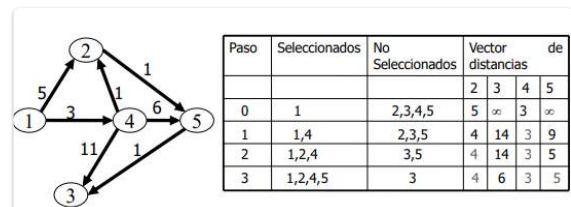
- Mantener un conjunto de nodos ya explorados para los cuales ya se ha determinado el camino más corto desde $v \rightarrow$ Algoritmo de Dijkstra
- Propiedades de los caminos mínimos: Si $d(v, u)$ es la longitud del camino mínimo para ir desde v a u , entonces se satisface $d(v, u) \leq d(v, s) + d(s, u)$

Datos relevantes:

- **Candidatos:** conjunto de vértices de los que se conoce la distancia mínima desde el origen
- **Solución:** cuando no quedan candidatos
- **Factibilidad:** siempre es factible
- **Objetivo:** minimizar el camino del origen al resto de nodos
- **Selección:** candidato con menor distancia al origen

Algoritmo de Dijkstra

Se parte de dos conjuntos de nodos ($V = S \cup C$), en el que S son los nodos seleccionados para los que se conoce el camino mínimo y C el resto de nodos del grafo. En cada iteración se escoge el nodo de C con menor distancia y se añade a S . Se recalcula los caminos a través del nodo seleccionado. Si no existe arista, se considera distancia infinita. Encuentra la solución óptima al problema.



```

def selectMinDistance(distances, visited):
    minDist = float('inf')
    bestItem = None
    for i in range(1, len(distances) + 1): # Iteramos por cada elemento del diccionario
        if not visited[i-1] and distances[i]['cost'] < minDist:
            minDist = distances[i]['cost'] # Guardamos la distancia mínima

```

```

        bestItem = i # Y la clave del nodo que no ha sido visitado
    return bestItem # Devolvemos la clave del nodo con menos coste que no haya sido visitado

def greedyDijkstra(origin, dest, g):
    size = len(g)
    node_info = {i: {'cost': float('inf'), 'pred': []} for i in range(1, size + 1)} #Por cada vertice del grafo
    guardo el coste de llegar hasta a él y una lista de predecesores para tener el path
    node_info[origin]['cost'] = 0 # El coste de llegar al origen logicamente es 0
    visited = [False] * size # Array de booleanos para los visitados suele ser mas eficiente
    visited[origin - 1] = True # Visitamos el origen
    for (start, end, weight) in g[origin]: # Por cada arista que parte del origen
        node_info[end]['cost'] = weight # Asignamos el peso al destino
        node_info[end]['pred'] += list([start]) # Añadimos su predecesor (origen)
    i = 1
    while i in range(1, size) and not visited[dest-1]: # Mientras no se haya visitado el destino (esto puede
    variar)
        nextNode = selectMinDistance(node_info, visited)
        if nextNode is not None: # Si hay algun candidato
            visited[nextNode - 1] = True # Lo visitamos
            for (start, end, weight) in g[nextNode]: # Por cada arista del nodo visitado
                cost = node_info[start]['cost'] + weight # Calculamos el coste de llegar del origen a él más
                el peso de dicha arista
                if cost < node_info[end]['cost']: # Si este es coste es menor que el que guarda el nodo destino
                de la arista
                    node_info[end]['cost'] = cost # Actualizamos el coste de la terminación de la arista
                    node_info[end]['pred'] = node_info[start]['pred'] + list([start]) # Asignamos a los
                    predecesores de la terminación de la arista los predecesores del nodo visitado más el nodo visitado en sí
                i += 1
    return node_info[dest]['cost'], node_info[dest]['pred'] + list([dest]) # Devolvemos el coste y el path del
    destino (esto puede variar dependiendo del problema)

#Dijkstra
g = {
    1: [(1,2,5), (1,4,3)],
    2: [(2,5,1)],
    3: [],
    4: [(4,2,1), (4,3,11), (4,5,6)],
    5: [(5,3,1)]
}

total, path = greedyDijkstra(1, 3, g)
print(f"Coste de llegar: {total}\nPath: {path}")

```

La complejidad de este algoritmo es de $O(|V|^2)$. Es recomendable utilizar una cola de prioridad (heap de Fibonacci) para una implementación eficiente que garantiza $O(|A| \log |V|)$.

Heurísticas voraces

Coloreado de grafos

🔗 Problema

Problema del coloreado de grafos: *Dado un grafo $G = (V, E)$ no dirigido, asignar un color a cada vértice de tal forma que dos adyacentes no tengan el mismo color.*

El objetivo es minimizar el número de colores utilizados. Sin embargo este es un **problema NP**, lo que implica que no existe ningún algoritmo eficiente que garantice un número mínimo de colores para grafos generales. Grafos planos: 4 colores (Teorema de Appel-Hanke)

Aquí se propone una solución eficiente no óptima en la que el orden en el que se escoja el nodo es decisivo.



Organiza la mejor fiesta universitaria

Organiza cualquier fiesta en CircusAlcala. Solo contáctanos y crearemos el evento perfecto para ti y tus amigos.



Viajante de comercio

🔗 Problema

Problema del viajante: *Dado un grafo $G = (V, E)$, encontrar un camino que empiece y acabe en un vértice v , pasando una única vez por cada vértice de V .*

El objetivo es obtener el circuito *hamiltoniano* de coste mínimo. Sin embargo este es un **problema NP**, lo que significa que no existe ningún algoritmo eficiente que garantice la optimalidad.

Para solucionarlo se emplean las heurísticas voraces:

- **Heurística 1:** escoger en cada iteración el vértice más cercano al último nodo añadido al circuito siempre que no se haya seleccionado previamente y no cierre el circuito.
- **Heurística 2:** escoger las aristas de coste mínimo (como en *Kruskal*) pero garantizando que al final se forme un circuito.

Escanea y únete a la fiesta en CircusAlcala. ¡Diversión asegurada en el mejor club!



WUOLAH