Centro de enseñanza técnica industrial

CETI

Tema: Árbol Parcial mínimo de Prim

Alumno: Isaac Carrillo Angulo

Reg. 21110358 grupo.6E1

Prof. Mauricio Alejandro Cabrera Arellano

Fecha: 24 de octubre de 2023

¿Qué es?

El algoritmo de Prim es un algoritmo perteneciente a la teoría de los grafos para encontrar un árbol recubridor mínimo en un grafo conexo, no dirigido y cuyas aristas están etiquetadas.

En otras palabras, el algoritmo encuentra un subconjunto de aristas que forman un árbol con todos los vértices, donde el peso total de todas las aristas en el árbol es el mínimo posible. Si el grafo no es conexo, entonces el algoritmo encontrará el árbol recubridor mínimo para uno de los componentes conexos que forman dicho grafo no conexo.

El algoritmo fue diseñado en 1930 por el matemático Vojtech Jarnik y luego de manera independiente por el científico computacional Robert C. Prim en 1957 y redescubierto por Dijkstra en 1959. Por esta razón, el algoritmo es también conocido como algoritmo DJP o algoritmo de Jarnik.

El algoritmo incrementa continuamente el tamaño de un árbol, comenzando por un vértice inicial al que se le van agregando sucesivamente vértices cuya distancia a los anteriores es mínima. Esto significa que en cada paso, las aristas a considerar son aquellas que inciden en vértices que ya pertenecen al árbol.

El árbol recubridor mínimo está completamente construido cuando no quedan más vértices por agregar.

El algoritmo podría ser informalmente descrito siguiendo los siguientes pasos:

Inicializar un árbol con un único vértice, elegido arbitrariamente, del grafo.

Aumentar el árbol por un lado. Llamamos lado a la unión entre dos vértices: de las posibles uniones que pueden conectar el árbol a los vértices que no están aún en el árbol, encontrar el lado de menor distancia y unirlo al árbol.

Repetir el paso 2 (hasta que todos los vértices pertenezcan al árbol)

Para hacerlo más en detalle, debe ser implementado el pseudocódigo siguiente.

Asociar con cada vértice v del grafo un número C[v] (el mínimo coste de conexión a v) y a un lado E[v] (el lado que provee esa conexión de mínimo coste). Para inicializar esos valores, se establecen todos los valores de C[v] a +∞ (o a cualquier número más grande que el máximo tamaño de lado) y establecemos cada E[v] a un valor "flag"(bandera) que indica que no hay ningún lado que conecte v a vértices más cercanos.

Inicializar un bosque vacío F y establecer Q vértices que aún no han sido incluidos en F (inicialmente, todos los vértices).

Repetir los siguientes pasos hasta que Q esté vacío:

Encontrar y eliminar un vértice v de Q teniendo el mínimo valor de C[v]

Añadir v a F y, si E[v] no tiene el valor especial de "flag", añadir también E[v] a F

Hacer un bucle sobre los lados vw conectando v a otros vértices w. Para cada lado, si w todavía pertenece a Q y vw tiene tamaño más pequeño que C[w], realizar los siguientes pasos:

Establecer C[w] al coste del lado vw

Establecer E[w] apuntando al lado vw

Devolver F

Como se ha descrito arriba, el vértice inicial para el algoritmo será elegido arbitrariamente, porque la primera iteración del bucle principal del algoritmo tendrá un número de vértices en Q que tendrán todos el mismo tamaño, y el algoritmo empezará automáticamente un nuevo árbol en F cuando complete un árbol de expansión a partir de cada vértice conectado del grafo. El algoritmo debe ser modificado para empezar con cualquier vértice particular s para configurar C[s] para que sea un número más pequeño que los otros valores de C (por norma, cero), y debe ser modificado para solo encontrar un único árbol de expansión y no un bosque entero de expansión, parando cuando encuentre otro vértice con "flag" que no tiene ningún lado asociado.

Hay diferentes variaciones del algoritmo que difieren unas de otras en cómo implementar Q: Como una única Lista enlazada o un vector de vértices, o como una estructura de datos organizada con una cola de prioridades, más compleja. Esta elección lidera las diferencias en complejidad de tiempo del algoritmo. En general, una cola de prioridades será más rápida encontrando el vértice v con el mínimo coste, pero ello conllevará actualizaciones más costosas cuando el valor de C[w] cambie.

¿Para qué sirve?

Una forma de implementar el algoritmo de Prim es así:

In [1]:

```
import numpy as np
import timeit
```

In [2]:

```
from graphs import NotOrientedGraph
from graphs import Vertex
```

El módulo graph contiene la estructura de datos utilizada para el grafo y la puedes encontrar en el repositorio de Nerve. El resto del algoritmo es:

In [3]:

```
def prim(graph: NotOrientedGraph, initial_vertex: Vertex) -> NotOrientedGraph:
    """
    Input:
```

```
    graph: It's a graph not oriented and connected.
    initial_vertex: It's one vertex from graph.
Output:
    The return is a graph not oriented and connected.
"""
# Initialize empty edges array and empty minimum spanning tree:
minimum_spanning_tree = dict()
visited_vertices = list()
edges = list()
min_edge = (None, None, np.infty)

# Arbitrarily choose initial vertex from graph:
vertex = initial_vertex

# Indices:
start_vertex, end_vertex, weight = range(3)

# Run prim's algorithm until we create an minimum spanning tree that
# contains every vertex from the graph:

try:
    while len(visited_vertices) < graph.num_vertices - 1:

        # Mark this vertex as visited:

        visited_vertices.append(vertex)

        # Set of potential edges:
        edges += vertex.edges

        # Find edge with the smallest weight to a vertex that has not yet
        # been visited:
        for edge in edges:
            inequality = edge[weight] < min_edge[weight]
            membership = edge[end_vertex] not in visited_vertices
            if inequality and membership:
                min_edge = edge

        # Get the start and end node from minimum edge:
        start_min_edge = min_edge[start_vertex]
        end_min_edge = min_edge[end_vertex]
        min_weight = min_edge[weight]

        # Add the minimum edge to minimum spanning tree:
```

```
            if minimum_spanning_tree.get(start_min_edge.id):
                edge = (end_min_edge.id, min_weight)
                minimum_spanning_tree[start_min_edge.id].append(edge)
            else:
                edge = [(end_min_edge.id, min_weight)]
                minimum_spanning_tree[start_min_edge.id] = edge

            # Remove min weight edge form list of edges:
            edges.remove(min_edge)

            # Start at new vertex and reset min edge:
            vertex = end_min_edge
            min_edge = (None, None, np.infty)

        except Exception as e:
            print('The graph is not connected or has no edges!')

        # Return the optimal tree:
        return NotOrientedGraph(minimum_spanning_tree)
```

Para usar el algoritmo de Prim es necesario tener un grafo no orientado que podemos construir de la siguiente manera:

In [4]:

```
data = {
    'a': [('c', 0)],
    'b': [('c', 1), ('e', 3)],
    'c': [('a', 3), ('b', 3), ('d', 2), ('e', 1)]
}
```

In [5]:

```
graph = NotOrientedGraph(data)
```

La clase `NotOrientedGraph` tiene varios métodos que permite consultar algunas de las propiedades de nuestro grafo, a continuación veamos algunos ejemplos:

In [6]:

```
graph.get_vertices()
```

Out[6]:

```
['a', 'c', 'b', 'e', 'd']
```

In [7]:

```
graph.get_edges()
```

Out[7]:

```
([('a', 'c', 0),
  ('c', 'a', 3),
  ('c', 'd', 2),
  ('c', 'b', 3),
```

```
 ('c', 'e', 1),
 ('b', 'e', 3)],
 ('b', 'c', 1),
 graphs.NotOrientedGraph)
```

In [8]:

```
graph.weight_matrix()
```

Out[8]:

```
(matrix([[inf,  0., inf, inf, inf],
        [ 3., inf,  3.,  1.,  2.],
         [inf, inf, inf, inf, inf],
        [inf,  1., inf,  3., inf],
 ['a', 'c', 'b', 'e', 'd'])
         [inf, inf, inf, inf, inf]]),
```

In [9]:

```
graph.adjacency_matrix()
```

Out[9]:

```
(matrix([[0, 1, 0, 0, 0],
        [1, 0, 1, 1, 1],
         [0, 0, 0, 0, 0],
        [0, 1, 0, 1, 0],
 ['a', 'c', 'b', 'e', 'd'])
         [0, 0, 0, 0, 0]]),
```

Además también se necesita de un vértice inicial que para nuestro ejemplo será:

In [10]:

```
initial_vertex = graph.get_vertex('a')
```

Al ejecutar el algoritmo de Prim se obtiene un nuevo grafo conexo y no orientado:

In [11]:

```
tree = prim(graph, initial_vertex)
```

Para nuestro ejemplo, se peude verificar que en efecto el grafo obtenido pasa por todos los vertices y además también nos entrega cuales son la arista que se usando para construirlo.

In [12]:

```
tree.get_vertices()
```

Out[12]:

```
['a', 'c', 'e', 'd', 'b']
```

In [13]:

```
tree.get_edges()
```

Out[13]:

```
([('a', 'c', 0), ('c', 'e', 1), ('c', 'd', 2), ('c', 'b', 3)],
 graphs.NotOrientedGraph)
```

¿Cómo se implementa en el mundo?

Una de las aplicaciones más destacadas del árbol mínimo recubridor se encuentra en el ámbito de las telecomunicaciones, por ejemplo, en las redes de comunicación eléctrica, telefónica, etc. Los nodos representarían puntos de consumo eléctrico, teléfonos, aeropuertos o computadoras. Las aristas podrían ser cables de alta tensión, fibra óptica, rutas aéreas,... .

¿Cómo lo implementarías en tu vida?

En la toma de decisiones, evaluación de opciones imagina que estás enfrentando una decisión importante que implica múltiples opciones con diferentes pros y contras. aplica el principio de Prim evaluando las "distancias" entre las opciones en términos de sus ventajas y desventajas. - priorización: utiliza el algoritmo de Prim para priorizar tus tareas diarias, enfocándote en las actividades que tienen el mayor impacto positivo en tu vida y eliminando las distracciones innecesarias. relaciones personales amistades y conexiones:

¿Cómo lo implementarías en tu trabajo o tu trabajo de ensueño?

Lo aplicaría por ejemplo para:

Redes de Computadoras: En la planificación de redes de computadoras, el algoritmo de Prim puede utilizarse para encontrar la infraestructura

Redes de Telecomunicaciones: Para la expansión de redes de telecomunicaciones, Prim puede ayudar a minimizar los costos de cableado al encontrar la disposición más

Diseño de Circuitos Electrónicos:

Diseño de Placas de Circuito Impreso: En el diseño de placas de circuito impreso (PCB)