

Parte 1

Ejercicio 1

Implementar la función crear grafo que dada una lista de vértices y una lista de aristas cree un grafo con la representación por Lista de Adyacencia.

```
1  def initialize(cls, nodes, key_pairs=[], dir_graph=False):
2      # vertices: List(GraphNode)
3      # key_pairs: List(node_key_0, node_key_1), adjacent nodes
4      # dir_graph: The graph could be constructed as directed
5
6      graph = Graph()
7      # Inserts all the nodes
8      for node in nodes:
9          graph.add(node)
10
11     # Creates the nodes link
12     if not dir_graph:
13
14         for (node_key0, node_key1) in key_pairs:
15             graph.link(node_key0, node_key1)
16     else:
17         for (node_key0, node_key1) in key_pairs:
18             graph.link_dir(node_key0, node_key1)
19     return graph
```

Todo el código básico relacionado con el grafo está implementado en una clase, donde `link(u, v)` es un método que crea la arista (u, v) y la (v, u) . De manera similar `link_dir(u, v)` solamente crea la arista (u, v)

Ejercicio 2

Implementar la función exists Path

```
1 def exists_path(graph, node0, node1):
2     """Returns if there is a path that connects the vertices"""
3     # Uses a similar algorithm to Breadth-First-Search
4
5     starting_node = graph[node0]
6     end_node = graph[node1]
7
8     queue = [starting_node]
9     traversed_keys = set()
10
11     while len(queue):
12         node = queue.pop(0)
13         if node.key == end_node.key:
14             return True
15         # Stores the traversed vertex in a dictionary
16         traversed_keys.add(node.key)
17         # Loops through all the adjacent vertices and adds them to the queue
18         for adjacent_node in graph.get_adjacent(node):
19             # If the vertex wasn't added to the queue, adds
20             if adjacent_node.key not in traversed_keys:
21                 queue.append(adjacent_node)
22
23     return False
```

Ejercicio 3

Implementación de una función que determina si un grafo es conexo

```
1 def is_connected(graph):
2     """
3     Uses a DFS approach. With DFS we can count the amount of
4     connected components in the main for loop. If the counter is
5     greater than 1, it means it's not connected
6     """
7     def _visit_node(graph, node, visited_keys):
8         # Recursive function, used to visit all the adjacent nodes
9         if node.key in visited_keys:
10             return
11
12         visited_keys.add(node.key)
13
14         for adjacent_node in graph.get_adjacent(node):
15             _visit_node(graph, adjacent_node, visited_keys)
16
17
18     visited_keys = set()
19     components = 0
20     for node in graph.nodes:
21
22         if node.key not in visited_keys:
23
24             # More than one connected component
25             if components == 1:
26                 return False
27
28             components += 1
29             _visit_node(graph, node, visited_keys)
30
31     return True
```

Ejercicio 4

Implementación de una función que determina si un grafo es un árbol, es decir que es **conexo y no tiene ciclos**

```

1  def is_tree(graph: Graph):
2      """
3      Uses a DFS approach.
4      Cycle detection: we can check if any of the adjacent nodes
5          have already been visited, and if it's not an immediate parent, it means
6          we found a regression edge -> cycles -> not tree
7      Connected: As in the is_connected(graph) implementation, counts the amount
8          of connected components, and in a tree it should be just one
9      """
10
11     def visit_is_tree_node(graph, node, visited_keys, immediate_parent):
12
13         # Node already visited
14         if node.key in visited_keys:
15             return True
16
17         # Visited for the first time
18         visited_keys.add(node.key)
19
20         # Loops through all the adjacent nodes
21         for adjacent_node in graph.get_adjacent(node):
22
23             if adjacent_node.key in visited_keys:
24
25                 # If the node is already visited and it's not its parent
26                 # it means we found a regression edge, making a cycle, then it's not a tree
27                 if adjacent_node != immediate_parent: return False
28
29             tree = visit_is_tree_node(graph, adjacent_node, visited_keys, node)
30             if not tree: return False
31
32         return True
33
34     # Uses hash table
35     visited_keys = set()
36     components = 0
37
38     for node in graph.nodes:
39
40         if node.key not in visited_keys:
41
42             # Not connected
43             if components == 1:
44                 return False
45
46             components += 1
47             tree = visit_is_tree_node(graph, node, visited_keys, None)
48
49             if not tree:
50                 return False
51
52     return True

```

Ejercicio 4

Función que determina si un grafo es completo, es decir que existe una arista para cada par de vértices. En la implementación asumo que el grafo es simple y que no hay aristas repetidas

```
1 def is_complete(graph):
2     """
3     A complete graph is an undirected graph in which every
4     pair of distinct vertices is connected by a unique edge
5     This implementation assumes that the graph is simple
6     """
7     for node in graph.nodes:
8
9         if len(graph) - 1 != len(list(graph.get_adjacent(node))):
10             return False
11
12     return True
```

Ejercicio 6

Implementar una función que dado un grafo devuelva una lista de aristas que si se eliminan el grafo se convierte en un árbol. Respetar la siguiente especificación.

```
1 def convert_tree(graph):
2     """
3     Returns the list of edges that when removed, make the graph a tree
4     Note it's almost the same algorithm as is_tree
5     """
6
7     def _visit(graph, node, visited_keys, remove_edges, immediate_parent):
8
9         if node.key in visited_keys:
10             return
11
12         visited_keys.add(node.key)
13
14         for adjacent_node in graph.get_adjacent(node):
15
16             if adjacent_node.key in visited_keys:
17
18                 if adjacent_node != immediate_parent:
19                     # Regression edge
20                     remove_edges.append((node, adjacent_node))
21
22             _visit(graph, adjacent_node, visited_keys, remove_edges, node)
23
24     remove_edges = []
25     visited_keys = set()
26
27     for node in graph.nodes:
28         _visit(graph, node, visited_keys, remove_edges, None)
29
30
31     return remove_edges
```

El problema se resuelve creando una lista con los arcos de retroceso

Parte 2

Ejercicio 7

Implementar una función que retorna la cantidad de componentes conexas

```
1 def connections_count(graph):
2
3     """
4     Returns the amount of connected components
5     Uses DFS, and counts the component in the main loop
6     """
7     def _visit(graph, node, visited_keys):
8         if node.key in visited_keys:
9             return
10
11         visited_keys.add(node.key)
12
13         for adjacent_node in graph.get_adjacent(node):
14             _visit(graph, adjacent_node, visited_keys)
15
16     visited_keys = set()
17     connections = 0
18
19     for node in graph.nodes:
20
21         if node.key not in visited_keys:
22             connections += 1
23             _visit(graph, node, visited_keys)
24
25     return connections
```

Ejercicio 8

Implementar una función que crea un árbol de expansión usando búsqueda en anchura

```
1 def breadth_first_search_tree(graph, root):
2     """
3     Returns a Tree
4     """
5
6     root_node = graph[root]
7
8     tree_graph = Graph()
9     tree_graph.add(root_node)
10
11     # Stores the keys of the traversed nodes
12     traversed_keys = set()
13     traversed_keys.add(root_node.key)
14     queue = [root_node]
15     while len(queue):
16         node = queue.pop(0)
17
18         for adjacent_node in graph.get_adjacent(node):
19
20             if adjacent_node.key not in traversed_keys:
21                 queue.append(adjacent_node)
22
23                 # Adds the node and links with it's parent
24                 tree_graph.add(adjacent_node)
25                 tree_graph.link(node, adjacent_node)
26                 traversed_keys.add(adjacent_node.key)
27
28     return tree_graph
```

Ejercicio 9

Implementar una función que crea un árbol de expansión usando búsqueda en profundidad

```
1 def depth_first_search_tree(graph : Graph, root):
2
3     tree_graph = Graph()
4     root_node = graph[root]
5
6     visited_keys = set()
7
8     time = 0
9
10    # Visits the root
11    dfs_visit(graph, tree_graph, root_node, time, visited_keys)
12
13    # Visits the other nodes
14    for node in graph.nodes:
15
16        # Visits the node
17        if node.key not in visited_keys:
18            dfs_visit(graph, tree_graph, node, time, visited_keys)
19
20    return tree_graph
21
22 def dfs_visit(graph, tree_graph, node, time, visited_keys):
23
24     tree_graph.add(node)
25     visited_keys.add(node.key)
26
27     time += 1
28     node.t0 = time
29
30     for adjacent_node in graph.get_adjacent(node):
31
32         # Backwards edge
33         if adjacent_node.key not in visited_keys:
34             dfs_visit(graph, tree_graph, adjacent_node, time, visited_keys)
35             tree_graph.link(node, adjacent_node)
36
37     time += 1
38     node.t1 = time
```


Ejercicio 10

Implementar una función que retorna el camino más corto entre dos vértices

```

1  def best_road(graph, start, end):
2      """
3      Returns a list of the edges that make the shortest path
4      Uses a BFS approach.
5      Stores all the posible paths inside a list, for each iteration,
6      a new path is added to paths, until we find the end
7      """
8
9      start_node = graph[start]
10     end_node = graph[end]
11
12     if start_node == end_node:
13         return [start_node]
14
15     visited_keys = set()
16     visited_keys.add(start_node.key)
17
18     paths = [[start_node]]
19     path_index = 0
20
21     while path_index < len(paths):
22
23         path = paths[path_index]
24         last_node = path[-1]
25
26         for adjacent_node in graph.get_adjacent(last_node):
27
28             # Already visited
29             if adjacent_node.key in visited_keys:
30                 continue
31
32             # Path found
33             if adjacent_node == end_node:
34                 return path + [adjacent_node]
35
36             visited_keys.add(last_node.key)
37             paths.append(path + [adjacent_node])
38
39         path_index += 1
40
41     return []
42

```

La implementación usa BFS, pero además guarda cada uno de los caminos. Una vez que se encuentra el nodo final se retorna el camino que llegó a ese nodo. Podemos asegurar

que es el más corto por la propiedad de que BFS siempre toma el camino más corto para cada par de vértices

Ejercicio 11 (Opcional)

Crear una función que determine si un grafo es bipartito

```

1  def is_bipartite(graph):
2
3      # A graph G=(V, E) is bipartite when for each edge
4      # (u, v) of E, u and v belong to different sets
5
6      def _visit_bipartite(graph, node, group_0, group_1, parent_group_0):
7
8          # 1) When the node is already added
9          if node.key in group_0:
10
11              # If the node is in the same group as the parent
12              if parent_group_0:
13                  return False
14
15              return True
16
17          if node.key in group_1:
18
19              # If the node is in the same group as the parent
20              if not parent_group_0:
21                  return False
22
23              return True
24
25          # 2) Adds the node to it's corresponding List (different to parent)
26          if parent_group_0:
27              group_1.add(node.key)
28
29          else:
30              group_0.add(node.key)
31
32          for adjacent_node in graph.get_adjacent(node):
33              bipartite = _visit_bipartite(graph, adjacent_node, group_0, group_1, not parent_group_0)
34              if not bipartite:
35                  return False
36
37          return True
38
39
40      group_0 = set()
41      group_1 = set()
42
43      for node in graph.nodes:
44
45          if node.key in group_0 or node.key in group_1:
46              continue
47          bipartite = _visit_bipartite(graph, node, group_0, group_1, False)
48          if not bipartite:
49              return False
50
51      return True

```

La idea del algoritmo anterior consiste en usar DFS, y en lugar de tener una única lista de nodos visitados tenemos 2. Además es necesario saber a qué grupo pertenece el nodo “parent” del actual, para poder agregar al nodo actual en otro grupo. De esta manera, cuando el nodo que visitamos ya fué agregado y además pertenece al mismo grupo que el padre podemos decir que no es bipartito (esto último se puede ver en los condicionales de `_visit_bipartite`)

Ejercicio 12

Demostrar que si el grafo G es un árbol y se le agrega una arista nueva entre cualquier par de vértices se forma exactamente un ciclo y deja de ser un árbol.

1)

\forall grafo G que sea un árbol.

Sean $u, v \in V$ dos vértices cualquiera de G

Si agregamos la arista (u, v) al árbol, entonces deja de ser un árbol

Demostración por contradicción: Supongo que la conclusión es falsa, por lo tanto

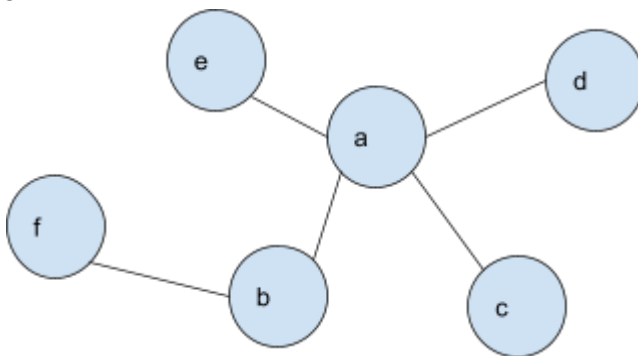
2)

\forall grafo G que sea un árbol.

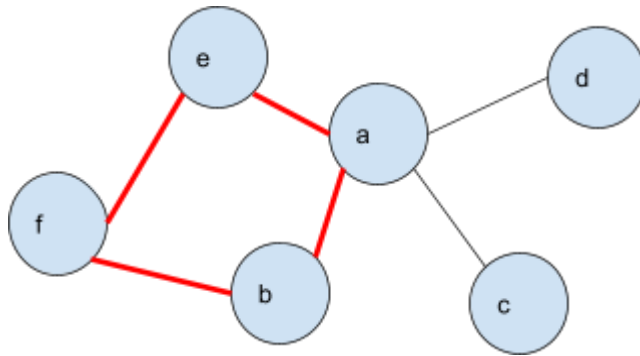
Sean $u, v \in V$ dos vértices cualquiera de G

Si agregamos la arista (u, v) al árbol, entonces el grafo sigue siendo un árbol

Sea $G = \{V = \{a, b, c, d, e, f\}, A = \{(a, b), (a, c), (a, d), (a, e), (b, f)\}\}$ el siguiente grafo. Se observa que es un árbol, pues no tiene ciclos



Agregando una arista para un par de vértices (e, f)



Vemos que se forma un ciclo, razón por la cuál G deja de ser un árbol. Esto nos muestra que el argumento 2) es falso.

Por lo tanto, queda demostrado que 1) es verdadero

Ejercicio 13

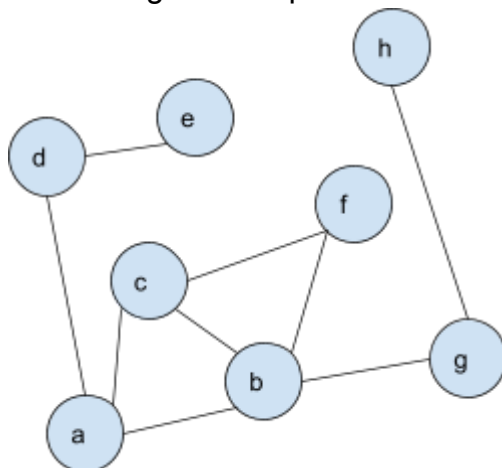
Demostrar que si la arista (u, v) no pertenece al árbol BFS, entonces los niveles de u y v difieren a lo sumo en 1.

Para todo árbol BFS $G = (V, A)$, si se toman dos vértices cualesquiera $v, u \in V$ tales que $(u, v) \notin A$.

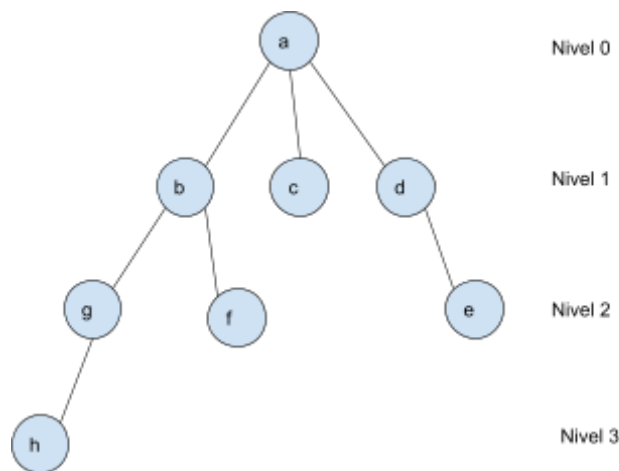
Los niveles de u y v difieren como máximo en 1

Creo que el enunciado es falso, por lo tanto busco un contraejemplo

Dada la siguiente representación de un grafo G



Su árbol de expansión generado por BFS considerando como prioridad al orden alfabético y tomando como raíz al vértice a resulta:



Se observa que $(h, a) \notin A$ y difieren en 3 niveles, por lo tanto queda demostrado por contraejemplo que el enunciado es **falso**

Ejercicio 13''

Interpreto "a lo sumo" como "por lo menos"

Para todo árbol BFS $G = (V, A)$, si se toman dos vértices cualesquiera $v, u \in V$ tales que $(u, v) \notin A$.

Los niveles de u y v difieren por lo menos en 1

Procedo a demostrar por contradicción, es decir

2) *Los niveles de u y v difieren como máximo en 1*

Gracias al punto anterior, sabemos que los niveles de u, v , pueden diferir en más de un nivel, por lo tanto 2) enunciado es falso, lo que hace que el enunciado principal sea **verdadero**

Ejercicio 14

Implementar el algoritmo de Prim para MST

```
1 def prim(graph: dict):
2
3     heap = Heap(len(graph))
4     smaller_weights = {}
5
6     # Initializes the heap and weights as maximum
7     for node in graph.keys():
8         min_weight = 1e7
9         smaller_weights[node] = min_weight
10        heap.add(node, min_weight)
11
12    # Updates the first key, so it gets extracted first
13    first_key = list(graph.keys())[0]
14    heap.update_key(first_key, 0)
15
16    # Holds the parent-child results
17    parent = {}
18
19    while len(heap):
20
21        # Gets the edge with the smallest weight
22        min_key, min_weight = heap.pop()
23
24        for (adjacent_node, weight) in graph[min_key]:
25
26            # If it doesn't make a cycle and the weight isn't the smallest
27            if adjacent_node in heap and weight < heap.access(adjacent_node):
28                parent[adjacent_node] = (min_key, weight)
29
30            # Adds to the tree
31            heap.update_key(adjacent_node, weight)
32
33    # Builds the graph, using the parent dict
34    result_tree = {}
35    total_weight = 0
36    for node in parent.keys():
37        child, weight = parent[node]
38        weight_insert_double(result_tree, node, child, weight)
39
40        total_weight += weight
41
42    print(f"Weight: {total_weight}")
43
44    return result_tree
```

La implementación anterior se basa en el uso de Min Heap.

1. Lo primero que hago es agregar todas las keys al Heap, donde el peso mínimo de cada una es inicializado con un número muy grande (supongamos que es infinito)
2. Luego, selecciono la primera key del diccionario, y le pongo como peso mínimo 0, de tal manera que en el siguiente bucle sea “popeado” primero
3. Parent es un diccionario, que dado un vértice se almacena su parent dentro del árbol mínimo de expansión
4. Si hay elementos restantes en heap, hay vértices que todavía no han sido agregados al árbol
5. Obtengo el vértice u , junto a su peso, el cuál resulta el menor de todo el grafo
6. Itero por todos los vértices adyacentes v . Si todavía no lo agregamos y el peso de la arista (u, v) es menor que el actual, entonces actualizo el valor en heap y vínculo con parent. Nótese que todavía no retiramos al vértice del heap, por lo cuál puede llegar a existir una arista de v donde su peso sea menor y su parent no termine siendo u
7. Finalmente, construyo el MST teniendo como datos al diccionario parent

Complejidad:

Se observa que el contenido del bucle for interior, el cuál itera por todos los vértices se va a ejecutar $O(|V| + |A|)$, pues con en while vamos a iterar por todos los vértices de V , y con el bucle for, vamos a terminar realizando $2|A|$ iteraciones.

Pero dentro del bucle, usamos a Min Heap. En el peor de los casos, siempre accedemos dentro del condicional if, haciendo que ejecutemos la operación `update_key`, la cuál tiene una complejidad temporal $O(\log(|V|))$.

Por lo tanto, la complejidad de mi algoritmo resulta $O((V + A)\log(V))$

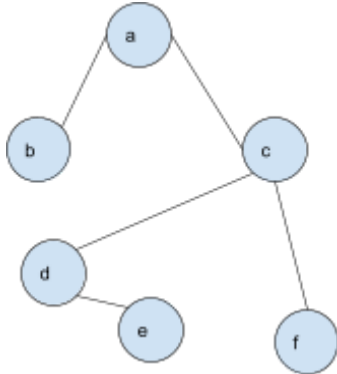
Ejercicio 15

Implementar el algoritmo de Kruskal para MST

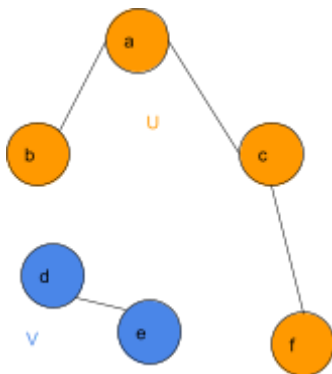
Ejercicio 16

Demostrar que si la arista (u,v) de costo mínimo tiene un nodo en U y otro en $V - U$, entonces la arista (u,v) pertenece a un árbol abarcador de costo mínimo.

Supongamos que tenemos un MST T del grafo $G = (V, A)$ y se ve de la siguiente manera



Si nosotros retiramos una arista cualquiera de T , vamos a obtener un bosque, formado por T_1 y T_2



Ahora, podemos observar lo que plantea el ejercicio, dos grupos de vértices U y V , donde tenemos una arista de costo mínimo (c, d) . Se observa que si agregamos la arista, es imposible que se formen ciclos, pues $c \in U$ y $d \in V$. Lo cuál demuestra que el enunciado es verdadero


Ejercicio 21

```
1 class Node:
2     def __init__(self, key) -> None:
3         self.key = key
4         self.d = float("inf")
5         self.parent = None
```

Etapas de inicialización

```
1
2 def dijkstra(graph, start_key, end_key):
3
4     weights = {}          # Stores the weight of each (u, v) edge
5     nodes = {}            # Stores the Node instances
6
7     edges_count = 0
8
9     # Creates the node instances, and fills the weights dict
10    for node_key in graph.keys():
11
12        # The node could be added here
13        nodes[node_key] = Node(node_key)
14
15        for ady, weight in graph[node_key]:
16            edges_count += 1
17            weights[(node_key, ady)] = weight
18
19    # Creates the heap and initializes d attribute as +infinite
20    heap = Heap(edges_count)
21    for node_key in graph.keys():
22        heap.add(node_key, float("inf"))
23
24    visited_keys = set()
25
26    # Sets the d value of the starting node to 0
27    # this way the heap pops the start first
28    heap.update_key(start_key, 0)
29    nodes[start_key].d = 0
```

Etapas de ejecución



```

1
2  while len(heap):
3
4      (node_key, d) = heap.pop()
5      visited_keys.add(node_key)
6
7      # Gets the node, containing d, and parent data
8      node = nodes[node_key]
9
10     for ady, weight in graph[node_key]:
11
12         if ady not in visited_keys:
13
14             ady_node = nodes[ady]
15
16             # Relax the adjacent
17             modified = relax(node, ady_node, weights)
18
19             # Only if the relax call modified the value
20             if modified:
21                 heap.update_key(ady, ady_node.d)
22
23 end_node = nodes[end_key]
24
25 # No path, unreachable from start_key
26 if end_node.parent is None:
27     return None
28
29 # Traverses the path backwards, using the parent attribute
30 path = []
31 parent = end_node
32 while parent is not None:
33     path.append(parent.key)
34     parent = parent.parent
35
36 return path
    
```

Funcionamiento

- Etapa de inicialización
 - `weights` es un diccionario que contiene el peso de toda arista (u, v)
 - `nodes` es un diccionario que dada la key de un vértice, retorna su nodo correspondiente. Nótese que los nodos guardan la información de d y de *parent*
 - `heap` es una *min heap*, usada como *cola* para ir retirando los nodos con menor d
 - Se observa que actualizo el valor $d = 0$ del nodo de entrada, o comienzo, para que sea el primero en ser retirado de `heap`
- Etapa de ejecución
 - Se retira el nodo con menor d
 - Se lo agrega al set de visitados
 - Para todo adyacente que no haya sido visitado:
 - Se lo “*relaja*”
 - Si el valor d del nodo adyacente fue modificado, entonces se actualiza en `heap`
- Etapa de finalización:
 - Si el nodo “*final*” no tiene `parent`, entonces “`start_key`” y “`end_key`” se encuentran en dos componentes conexas disjuntas del grafo -> None
 - Caso contrario, existe un camino. Se recorre el camino de `parents`, el cuál resulta ser el mejor camino y se lo retorna

Complejidad

- En la etapa de inicialización, tenemos una complejidad $O(V)$, pues iteramos por los vértices/nodos dos veces.
- En la etapa de ejecución, mediante la estructura `heap`, vamos a iterar por todos los vértices del grafo, y para cada uno de estos vamos a iterar por todos sus adyacentes. Es decir $O(V + A)$, pero observamos que en el peor caso, siempre vamos a tener que actualizar el valor d de los nodos, una operación de orden $O(\log(V))$. Razón por la cuál, la complejidad temporal en esta etapa es de $O((V + A)\log(V))$
- En la etapa de finalización, en el peor de los casos, vamos a tener que recorrer $|V|$ nodos buscando el camino más corto, lo cuál hace $O(V)$

Claramente la complejidad resulta determinada por la etapa de ejecución, resultando

$$O((V + A)\log(V))$$