

# Algoritmos y Estructuras de Datos II

TALLER - 8 de Junio de 2021

## Laboratorio 7: Algoritmo de Dijkstra

- Revisión 2021: Marco Rocchietti  
- Revisión 2020: Leonardo Rodríguez  
(:)

### Objetivos

1. Representar un Grafo en C
2. Reforzar conceptos de memoria estática (STACK) y memoria dinámica (HEAP)
3. Manejar vectorización de matrices
4. Construir matrices dinámicas en C
5. Implementar el algoritmo de Dijkstra

### Preliminares

En este laboratorio se deberá implementar el algoritmo de *Dijkstra*. El algoritmo computa, dado un vértice inicial  $v$ , los costos de los caminos mínimos entre  $v$  y cada uno de los demás vértices de un grafo dirigido. Nos vamos a basar en el pseudocódigo visto en el teórico, donde se define una función

```
fun Dijkstra (L : array[1..n,1..n] of Nat, v: Nat) ret D : array [1..n] of Nat  
  (:)  
end fun
```

Nuestra versión en C tendrá algunas pequeñas diferencias. Por ejemplo, no se tomará como entrada una matriz sino un TAD *Grafo*. La función que implementará el algoritmo se encuentra en el archivo `dijkstra.c` y posee la siguiente signatura:

```
cost_t *dijkstra(graph_t graph, vertex_t init);
```

El primer parámetro es un grafo, y el segundo es el vértice inicial. Devuelve un arreglo que en cada posición  $v$  (el tipo `vertex_t` es alguna variedad de enteros y se puede utilizar para indexar) contiene el costo del camino de costo mínimo desde `init` hasta  $v$ .

Además se provee en `graph.h` una especificación del TAD Grafo (grafos dirigidos) y en `mini_set.h` una especificación del TAD *Set* con las operaciones relevantes de conjuntos que se necesitan para implementar el algoritmo de Dijkstra. Por último, el tipo para los costos de las aristas está especificado en `cost.h`.

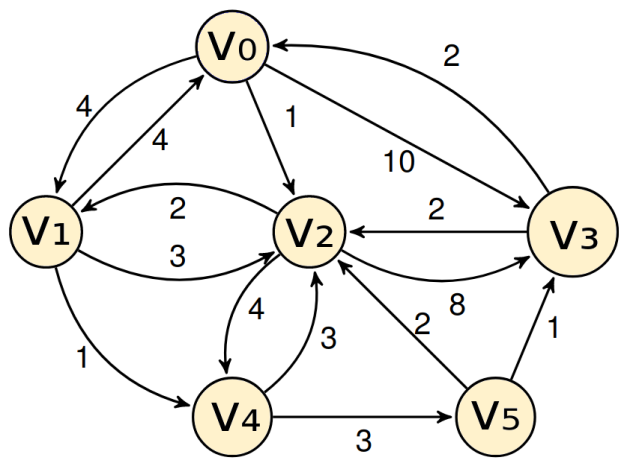
Se incluye un `Makefile` para que puedan compilar los distintos ejercicios de manera cómoda.

# Ejercicio 1: Representación de Grafos

El TAD grafo tiene la siguiente interfaz:

Función	Descripción
<code>graph_t graph_empty(unsigned int max_vertices)</code>	Crea un grafo vacío con capacidad máxima de <code>max_vertices</code> vértices
<code>unsigned int graph_max_vertices(graph_t graph);</code>	Devuelve la capacidad máxima del grafo
<code>void graph_add_edge(graph_t graph, vertex_t from, vertex_t to, cost_t cost)</code>	Agrega una arista desde el vértice <code>from</code> hasta <code>to</code> con costo <code>cost</code> .
<code>cost_t graph_get_cost(graph_t graph, vertex_t from, vertex_t to)</code>	Se obtiene el costo de la arista que hay de <code>from</code> a <code>to</code>
<code>graph_t graph_from_file(const char *file_path)</code>	Lee un grafo del archivo <code>file_path</code>
<code>void graph_print(graph_t graph);</code>	Muestra un grafo por pantalla
<code>graph_t graph_destroy(graph_t graph)</code>	Destruye la instancia <code>graph</code>

En la carpeta `input` se encuentran archivos en formato texto que especifican una matriz de costos. Por ejemplo el grafo dado en las filminas del teórico:



se representa en el archivo `input/example_graph_1.in` con la siguiente matriz:

6
0 4 1 10 # #
4 0 3 # 1 #
# 2 0 8 4 #
2 # 2 0 # #
# # 3 # 0 3
# # 2 1 # 0

El primer número es la cantidad de filas y columnas (es una matriz cuadrada por lo que

sólo un número es suficiente). Los vértices del grafo son  $\{0, 1, 2, 3, 4, 5\}$ , el costo de la arista (2, 3) es 8. Cuando no existe la arista, se escribe # para representar el costo infinito.

En el archivo `cost.h` van a encontrar la abstracción para los costos de las aristas.

a) Analizar la implementación parcial del TAD Grafo en `graph_bad.c`. Compilar mediante

```
$ make graph_bad
```

y luego ejecutar

```
$ ./graph_bad input/example_graph_3.in
```

Si no notan nada raro ejecuten

```
$ make test_bad
```

para ver qué dice `valgrind`. ¿Cuál es el problema de esta implementación?

Crear el archivo `graph.c` con la implementación de `graph_bad.c` corregida. ¿Hay alguna invariante de representación para definir?

b) Completar el archivo `graph.c` que utiliza la siguiente estructura de representación:

```
struct graph_data {
    cost_t **costs;
    unsigned int max_vertexs;
};
```

manejar correctamente la matriz de adyacencias y asegurarse de que no haya problemas de memoria. Actualizar la invariante de representación.

## Ejercicio 2: Dijkstra

Implementar el algoritmo de *Dijkstra* en `dijkstra.c` usando como guía el pseudocódigo dado en el teórico práctico:

```
fun Dijkstra (L : array[1..n,1..n] of Nat, v: Nat) ret D : array [1..n] of Nat
  var c : Nat
  var C : Set of Nat
  for i := 1 to n do add(C,i) od
  elim(C,v)
  for j := 1 to n do D[j] := L[v,j] od
  do (not is_empty_set(C)) →
    c := "elijo elemento c de C tal que D[c] sea mínimo"
    elim(C, c)
    for j in C do D[j] := min(D[j], D[c] + L[c,j]) od
  od
end fun
```

Van a necesitar copiar alguna de las versiones de `graph.c` para que funcione. Una vez terminado pueden compilar usando el `Makefile`.

Para verificar su buen funcionamiento, al ejecutar

```
$ ./dijkstra input/example_graph_1.in
```

Deberían obtener el siguiente resultado:

```
Dijkstra Shortest Path Algorithm  
Minimum cost from 0 to 0: 0  
Minimum cost from 0 to 1: 3  
Minimum cost from 0 to 2: 1  
Minimum cost from 0 to 3: 8  
Minimum cost from 0 to 4: 4  
Minimum cost from 0 to 5: 7
```

que se corresponde con la solución vista en el teórico:

