

TRABAJO PRÁCTICO 2

TEORÍA DE ALGORITMOS

(75.29 / 95.06 / TB024)

Cuatrimestre: 2º Cuatrimestre 2024

Fecha de entrega: 03 / 11 / 2024

Grupo: 5 (She Don't Give Grafo)

Curso: 03 - Echevarría

Integrantes:

Pablo Choconi	106388
Valentín Savarese	107640
Gian Luca Spagnolo	108072
Brian Céspedes	108219
Néstor Palavecino	108244

Índice

1. PROBLEMA 1.....	3
1.1. Enunciado y Supuestos.....	3
1.2. Diseño.....	3
1.3. Pseudocódigo.....	4
1.4. Implementación.....	5
1.5. Seguimiento.....	6
1.6. Complejidades.....	7
1.7. Sets de Datos y Tiempos de Ejecución.....	8
2. PROBLEMA 2.....	11
2.1. Enunciado y Premisas.....	11
2.2. Variables Utilizadas y Función Objetivo.....	12
2.3. Restricciones del Modelo.....	12
2.4. Software Utilizado y Resultados Obtenidos.....	13
2.5. Tabla de Resultados.....	13
2.6. Informe de la Solución.....	13
3. PROBLEMA 3.....	14
3.1. Enunciado y Supuestos.....	14
3.2. Análisis del Algoritmo.....	14
3.3. Visualización con Datos de Prueba.....	15
3.4. Estructuras de Datos.....	16
3.5. Complejidad Temporal y Espacial.....	17
3.6. Errores Detectados.....	17
3.7. Posibles Mejoras.....	21
REFERENCIAS.....	22

1. PROBLEMA 1

1.1. Enunciado y Supuestos

Usted trabaja en un equipo de consultoría de software en la Facultad de Ingeniería. Cada semana $1 \leq i \leq n$ debe seleccionar un tipo de trabajo para que realice el equipo. Los tipos de trabajo se pueden clasificar en "*tranquilos*" (por ejemplo, levantar un micrositio web para alguna secretaría de la facultad) y "*estresantes*" (por ejemplo, ayudar a un grupo de alumnos de TDA a completar la entrega en fecha de un TP). Si en la semana $i > 1$ se decide hacer un trabajo "*estresante*", entonces en la semana $i - 1$ no se puede hacer ningún trabajo (el equipo debe descansar). Los trabajos "*tranquilos*" no requieren descanso previo, e incluso se puede realizar un trabajo "*tranquilo*" la semana después de haber realizado un trabajo "*estresante*". Para $i = 1$, es posible tomar un trabajo "*estresante*" (suponemos que el equipo está descansado). Un trabajo "*tranquilo*" realizado en la semana i reportará un beneficio t_i , mientras que un trabajo "*estresante*" reportará un beneficio e_i ($t_i, e_i > 0$). Los trabajos duran exactamente una semana. Dado un conjunto de valores $\{t_1, t_2, \dots, t_n\}$ y $\{e_1, e_2, \dots, e_n\}$, determinar una planificación de trabajos que maximice el beneficio total obtenido.

Se pide:

- Diseñar e implementar un algoritmo de **Programación Dinámica** que determine el beneficio total óptimo e indique cuál es la secuencia de tareas a realizar.
- También, tenemos en cuenta los siguientes supuestos:
 - Los primeros índices (índice 0) de las listas de trabajos tranquilos y trabajos estresantes tienen el valor 0 (y no se tienen en cuenta) para simplificar el algoritmo, ya que i toma valores de 1 a n . Esto no quiere decir que las listas de entrada deban contener ese 0 inicial, ya que nuestro algoritmo se encarga de agregarlo.

1.2. Diseño

Para este ejercicio, se plantea un algoritmo de *programación dinámica* el cual hace uso de todas las bondades de esta metodología de resolución de problemas. Formaremos iterativamente la solución sin tener que hacer cálculos redundantes a medida que avanzamos.

La solución se basa en realizar los trabajos, ya sean tranquilos o estresantes en cada iteración. Siempre nos guiamos en que tenemos que tener particular cuidado con los trabajos estresantes. Esto es debido a que hacer un trabajo estresante implica no trabajar la semana anterior, así que tendremos que verificar que esta decisión realmente valga la pena.

- **Ecuación de recurrencia:**

$$\text{opt}(n) = \begin{cases} \max(\text{trabajos_t}[1], \text{trabajos_e}[1]) & \text{Si } i = 1 \\ \max(\text{trabajos_t}[1] + \text{trabajos_t}[2], \text{trabajos_e}[2]) & \text{Si } i = 2 \\ \max(\text{trabajos_t}[i - 1] + \text{trabajos_t}[i], \text{trabajos_e}[i - 2] + \text{trabajos_e}[i]) & \text{Si } i > 2 \end{cases}$$

- Se respeta el requisito de la subestructura óptima ya que en cada iteración estamos utilizando soluciones óptimas de problemas que ya hemos resuelto previamente

(subproblemas), esto para llegar finalmente al óptimo global, En otras palabras, la solución óptima global también contiene a su vez, la solución óptima de todos los subproblemas.

- Se cumple el requisito de superposición de problemas ya que en cada iteración vuelven a aparecer soluciones de subproblemas ya resueltos, los cuales reutilizamos (no los volvemos a calcular).
- Se utilizarán tres listas: una lista correspondiente al beneficio de cada trabajo tranquilo, y una lista correspondiente al beneficio de cada trabajo estresante, ordenados por semanas. Además, tendremos una lista de seleccionados para indicar qué trabajos realizaremos. A priori, también se podría utilizar una cuarta lista para almacenar los óptimos parciales de cada semana, pero en nuestra implementación se omitió esto, para optimizar recursos.

1.3. Pseudocódigo

elegir_trabajo(opt_actual, opt_anterior, i, seleccionados, trabajos_tranquilos, trabajos_estresantes):

```
# Tenemos dos alternativas:
# El óptimo es la suma del óptimo de la semana anterior + el trabajo tranquilo.
# El óptimo es la suma del óptimo de hace 2 semanas + el trabajo estresante.
opt = max(opt_actual + trabajos_tranquilos[i], opt_anterior + trabajos_estresantes[i])

SI opt == trabajos_tranquilos[i] + opt_actual:
    AGREGAR "t{i}" a seleccionados
SINO:
    AGREGAR "e{i}" a seleccionados

opt_anterior = opt_actual
opt_actual = opt

RETORNAR opt_actual, opt_anterior
```

seleccion_de_trabajos(trabajos_tranquilos, trabajos_estresantes):

```
AÑADIR 0 en trabajos_tranquilos en el índice 0
AÑADIR 0 en trabajos_estresantes en el índice 0

n = longitud(trabajos_tranquilos)
seleccionados = lista con un 0

# Primero calculamos el óptimo de las primeras dos semanas:
# En la función elegir_trabajo_segunda_semana() también se obtiene el óptimo de la primera semana.
opt_actual, opt_anterior = elegir_trabajo_segunda_semana(seleccionados, trabajos_tranquilos, trabajos_estresantes)

DESDE i = 3 HASTA n:
    opt_actual, opt_anterior = elegir_trabajo(opt_actual, opt_anterior, i, seleccionados,
        trabajos_tranquilos, trabajos_estresantes)
    seleccionados = ajustar_lista_seleccionados(seleccionados)

RETORNAR opt_actual, seleccionados
```

1.4. Implementación

```
def elegir_trabajo_primera_semana(seleccionados: list, trabajos_tranquilos: list,
trabajos_estresantes: list) -> int:

    opt_actual: int = max(trabajos_tranquilos[1], trabajos_estresantes[1])
    if opt_actual == trabajos_tranquilos[1]:
        seleccionados.append("t1")
    else:
        seleccionados.append("e1")

    return opt_actual

def elegir_trabajo_segunda_semana(seleccionados: list, trabajos_tranquilos: list,
trabajos_estresantes: list) -> tuple:

    opt_anterior: int = elegir_trabajo_primera_semana(seleccionados, trabajos_tranquilos,
trabajos_estresantes)
    opt_actual: int = max(opt_anterior + trabajos_tranquilos[2], trabajos_estresantes[2])
    if opt_actual == trabajos_tranquilos[2] + opt_anterior:
        seleccionados.append("t2")
    else:
        seleccionados.append("e2")

    return opt_actual, opt_anterior

def elegir_trabajo(opt_actual: int, opt_anterior: int, i: int, seleccionados: list,
trabajos_tranquilos: list, trabajos_estresantes: list) -> tuple:

    opt = max(opt_actual + trabajos_tranquilos[i], opt_anterior + trabajos_estresantes[i])
    if opt == trabajos_tranquilos[i] + opt_actual:
        seleccionados.append(f"t{i}")
    else:
        seleccionados.append(f"e{i}")

    opt_anterior = opt_actual
    opt_actual = opt

    return opt_actual, opt_anterior
```

```
def ajustar_lista_seleccionados(seleccionados: list) -> list:

    aux: list = []
    i: int = len(seleccionados) - 1

    # Recorro la lista de fin a principio, si encuentro un trabajo estresante, ignoro el
    # inmediatamente anterior y continuo iterando.
    while i > 0:
        aux.append(seleccionados[i])
        if "e" == seleccionados[i][0]:
            i -= 1
        i -= 1
    aux.reverse()

    return aux

def seleccion_de_trabajos(trabajos_estresantes: list, trabajos_tranquilos: list) -> tuple:

    # Agregó un 0 en el índice 0 para "descartarlo" (usaremos a partir del índice 1 para
    # simplificar).
    trabajos_tranquilos.insert(0, 0)
    trabajos_estresantes.insert(0, 0)

    n: int = len(trabajos_tranquilos)
    seleccionados: list = [0]

    opt_actual, opt_anterior = elegir_trabajo_segunda_semana(seleccionados, trabajos_tranquilos,
                                                              trabajos_estresantes)

    for i in range(3, n):
        opt_actual, opt_anterior = elegir_trabajo(opt_actual, opt_anterior, i, seleccionados,
                                                  trabajos_tranquilos, trabajos_estresantes)
        seleccionados = ajustar_lista_seleccionados(seleccionados)

    return opt_actual, seleccionados
```

1.5. Seguimiento

El algoritmo consiste en “recorrer” ambas listas de trabajos al mismo tiempo. En cada iteración verificaremos cuál es la alternativa que nos garantiza la optimalidad en ese instante. Si agregamos un trabajo estresante, significa que deberemos remover de la planificación el trabajo de la semana anterior, para garantizar el descanso.

De esta forma, nuestro algoritmo busca soluciones óptimas locales y también tiene un enfoque “bottom-up” propio de la programación dinámica, ya que empezamos a armar la planificación de los trabajos de manera incremental y desde cierto caso base.

A continuación, se verá un ejemplo paso a paso con un par de listas de trabajos en específico:

Lista de trabajos tranquilos = [8, 6, 4, 5, 2]

Lista de trabajos estresantes = [8, 6, 20, 2, 9]

- En la primera semana trataremos de elegir analizando el valor máximo entre 8 (t_1) y 8 (e_1). En esta situación, elegir cualquiera de las dos opciones en principio es lo mismo. Nuestro algoritmo siempre priorizará elegir los trabajos tranquilos. Si bien la ganancia es la misma, esto nos garantiza que potencialmente podremos realizar más trabajos.

Trabajo elegido: t_1 . Beneficio parcial: 8.

- En la segunda semana debemos elegir analizando el valor máximo entre $8 + 6 = 14$ ($t_1 + t_2$) y 6 (e_2 y descarto el trabajo de la semana 1 ya que se requiere descansar). En este caso, nos conviene hacer el trabajo t_2 ya que nos da un beneficio parcial de 14.

Trabajo elegido: t_2 . Beneficio parcial: 14.

- En la tercera semana debemos elegir analizando el valor máximo entre $14 + 4 = 18$ (la solución parcial anterior + t_3) y $8 + 20 = 28$ (la solución parcial de la primera semana + e_3 , descartando el trabajo en la semana 2 ya que se requiere descansar). En este caso, lo más conveniente es hacer el trabajo e_3 , obtendremos un beneficio parcial de 28, que es mejor que los 18 de beneficio que nos ofrece la otra alternativa.

○ **Trabajo elegido: e_3 (descartamos t_2). Beneficio parcial: 28.**

- En la cuarta semana debemos elegir analizando el valor máximo entre $28 + 5 = 33$ (la solución parcial anterior + t_4) y $14 + 2 = 16$ (la solución parcial de la segunda semana + e_4 , descartando el trabajo en la semana 3 ya que se requiere descansar). En este caso, nos conviene hacer el trabajo t_4 ya que nos garantiza un mayor beneficio parcial.

○ **Trabajo elegido: t_4 . Beneficio parcial: 33.**

- En la quinta y última semana debemos elegir analizando el valor máximo entre $33 + 2 = 35$ (la solución parcial anterior + t_5) y $28 + 9 = 37$ (la solución parcial de la tercera semana + e_5 , descartando el trabajo en la semana 4 ya que se requiere descansar). En este caso, haremos el trabajo e_5 .

Trabajos elegidos: [t_1 , e_3 , e_5]. Beneficio total: 37.

1.6. Complejidades

La *complejidad temporal* del algoritmo se verá principalmente influenciada por la función `selección_de_trabajos()`.

- Analizando la función `elegir_trabajo_primera_semana()`, la misma realiza operaciones para encontrar el máximo entre dos valores, y luego un append en una lista. Todo esto anterior constituye una complejidad $O(1)$.
 - **Complejidad resultante: $O(1)$**

- Analizando la función `elegir_trabajo_segunda_semana()`, el funcionamiento es muy similar a la función anterior. Llamar a la función `elegir_trabajo_primera_semana()` (analizada anteriormente) tiene una complejidad de $O(1)$. Por lo tanto, la complejidad de esta función también es $O(1)$.
 - **Complejidad resultante:** $O(1)$
- Analizando la función `elegir_trabajo()`, el funcionamiento es muy similar a las dos funciones anteriores, se calcula el máximo entre dos valores y se realizan inserciones sobre el final de una lista. Entonces, podemos afirmar que la complejidad de esta función nuevamente es $O(1)$.
 - **Complejidad resultante:** $O(1)$
- Analizando la función `ajustar_lista_seleccionados()` la misma itera sobre la lista de seleccionados, que tiene k elementos (como mucho $k = n$). Dentro del ciclo while, todas las operaciones son $O(1)$. El método `reverse()` tiene una complejidad de $O(m)$ siendo m la cantidad de elementos de la lista aux (como mucho, $m = n$).
 - **Complejidad resultante:** $O(n) + O(n) = O(n)$
- Analizando la función `seleccion_de_trabajos()`, inicialmente debemos colocar un 0 en los primeros índices de las listas (que en principio, son de la misma longitud), esto tiene una complejidad de $2 \cdot O(n) = O(n)$. Calcular la longitud de la lista y llamar a la función `elegir_trabajo_segunda_semana()` tiene una complejidad de $O(1)$. Luego, tenemos un ciclo **for** el cual itera $n - 3$ veces. En cada iteración, llamamos a la función `elegir_trabajo()` la cual ya analizamos que tiene una complejidad de $O(1)$. Finalmente llamamos a la función `ajustar_lista_seleccionados()` analizada previamente, la cual vimos que tiene una complejidad $O(n)$.

Complejidad final (temporal): $O(n) + O(1) + O(n) \cdot O(1) + O(n) = O(n)$

La **complejidad espacial**, por su parte, dependerá de la cantidad de trabajos queelijamos hacer. En el peor caso, haremos todos los trabajos tranquilos (n trabajos). Finalmente, el resultado en este caso será que tendremos una lista **seleccionados** de n elementos.

Complejidad final (espacial): $O(n)$

1.7. Sets de Datos y Tiempos de Ejecución

Para este ejercicio, se dispone del archivo `ejercicio_1_test.py`, el cual tiene las pruebas correspondientes para la ejecución de la implementación. En primera instancia, recorre las funciones pertenecientes al archivo `ejercicio_1.py` las cuales ejecutan la resolución de este ejercicio sin mantener una lista de óptimos, con el objetivo de ahorrar espacio al no utilizar estructuras adicionales, reduciendo así la **Complejidad Espacial**. Luego, ejecuta los mismos conjuntos de datos con aquellas funciones presentes en el archivo `ejercicio_1_con_lista_de_opt.py` donde la resolución del ejercicio mantiene una lista de óptimos.

En nuestro caso, la implementación que utiliza la *lista de óptimos* es simplemente anecdótica y para presentar el primer acercamiento que tuvimos al problema. Es importante

destacar que este acercamiento no solo nos brinda información sobre el óptimo global, sino que nos da una idea de cómo se forma la solución iterativamente, ya que en cada índice se encuentra el beneficio parcial obtenido en la semana *i*. La solución *sin lista de óptimos* consideramos que es superadora, principalmente porque está optimizada al no utilizar espacio en memoria adicional, resultando así en nuestra implementación definitiva de nuestro algoritmo.

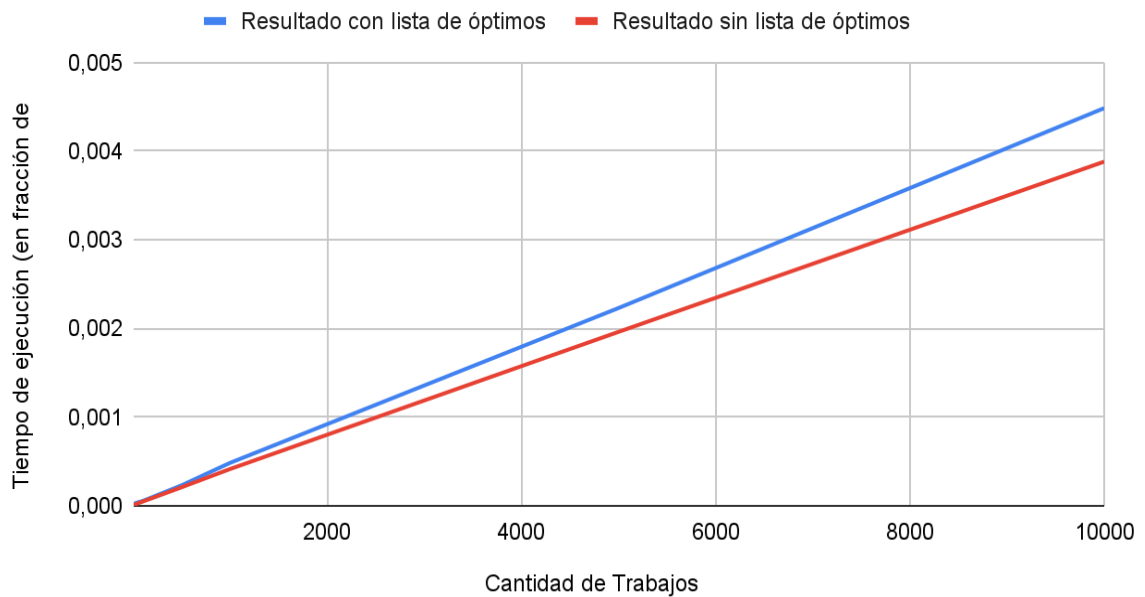
Con respecto a los **sets de datos** utilizados, se tienen dos listas a disposición: una de `trabajos_tranquilos` y otra de `trabajos_estresantes`, las cuales son leídas mediante archivos .csv. El resultado, de igual manera, es entregado como una lista de `trabajos_a_realizar` conteniendo la disposición de trabajos definida por nuestra implementación del ejercicio.

Teniendo en cuenta las implementaciones, se demuestra el siguiente gráfico el cual compara múltiples conjuntos de datos de hasta $n = 10.000$, para poder visualizar efectivamente como el tiempo de ejecución sube de forma lineal, de acuerdo a lo calculado anteriormente y a nuestra implementación definitiva del algoritmo *sin lista de óptimos*:



Asimismo, en base a lo explicado anteriormente, se dispone del siguiente gráfico el cual muestra específicamente la breve pero existente diferencia en tiempos de ejecución al mantener una *lista de óptimos* o no, teniendo en cuenta la explicación anterior de ambas implementaciones:

Planificación de trabajos (comparación con lista de óptimos)



Nosotros elegimos el valor final de $n = 10.000$ para poder visualizar correctamente la **Complejidad Temporal** correspondiente de nuestra implementación, y afirmar que coincide con nuestros cálculos, resultando en $O(n)$.

De igual manera, acordamos que este valor de n es suficiente como para poder visualizar una diferencia entre la implementación *con lista de óptimos* y *sin lista de óptimos*.

- Ocupa 100 m².
- Aporta un ingreso adicional de \$15.000.000.
- *La temporada dura un mes.*

2.2. Variables Utilizadas y Función Objetivo

- x_1 = Cantidad de carpas
- x_2 = Cantidad de toldos
- x_3 = Cantidad de sombrillas
- x_4 = Incluir bar (variable binaria)

La función a maximizar es la siguiente:

$$z = 3000000 \cdot x_1 + 1000000 \cdot x_2 + 500000 \cdot x_3 + 15000000 \cdot x_4$$
$$\max(z)$$

2.3. Restricciones del Modelo

- Las inecuaciones que delimitan los valores permitidos son:

$$x_1 \leq 500$$

$$x_2 \leq 200$$

$$10 \cdot x_1 + 5 \cdot x_2 + 5 \cdot x_3 + 100 \cdot x_4 \leq 6600$$

$$x_1 - 2 \cdot x_2 = 0$$

- La *primera ecuación* nos indica que no se pueden instalar más de 500 carpas en el balneario.
- La *segunda ecuación* nos indica que no se pueden instalar más de 200 toldos en el balneario.
- La *tercera ecuación* nos indica que la suma de los m² de todas las carpas, con los m² de todos los toldos, con los m² de todas las sombrillas, con los m² del bar (ya sea que éste se incluya o no) no puede superar el espacio disponible en el balneario para ubicar cualquiera de los 4 tipos de elementos mencionados.
 - Los coeficientes representan los m² de cada elemento.
- El espacio disponible en el balneario resulta de a los 10.000 m² originales restarle los 3.000 m² de recreación y los 400 m² de la administración, vestuarios, duchas y baños.
- Los coeficientes para x_2 y x_3 son 5 porque las carpas ocupan el doble de espacio que los toldos y éstos últimos igual espacio que cada sombrilla. Y como las carpas ocupan 10 m², entonces tanto los toldos como las sombrillas ocupan 5 m² de espacio.
- La *cuarta ecuación* nos indica que la demanda de carpas será el doble de la demanda de toldos.

2.4. Software Utilizado y Resultados Obtenidos

El software utilizado para resolver el problema es **LINDO**, un programa de optimización usado para resolver problemas de programación lineal.

Los resultados obtenidos están dentro de la carpeta `./ej2/res`.

- La solución óptima que nos da LINDO es la siguiente:

- x_1 (carpas) = 400
- x_2 (toldos) = 200
- x_3 (sombrillas) = 300
- x_4 (bar) = 1

Esta configuración entre el bar, las carpas, las sombrillas y los toldos nos dará una ganancia máxima de **\$1.565.000.000**.

En la carpeta `./ej2` se incluye el archivo `Ejercicio 2 PL.ltx` que es un script de LINDO que contiene la función a maximizar y las restricciones del modelo de PL.

2.5. Tabla de Resultados

Variable	Descripción	Valor obtenido
x1	Cantidad de carpas	400
x2	Cantidad de toldos	200
x3	Cantidad de sombrillas	300
x4	Booleano de incluir el bar (0 = no, 1 = sí)	1

2.6. Informe de la Solución

Lo que la resolución del modelo de **Programación Lineal** nos propone como solución a nuestro problema, es que si queremos maximizar la facturación para la próxima temporada de verano, lógicamente ocupando el mayor espacio posible de nuestro balneario, es la siguiente:

- Se deberán utilizar 400 carpas (4000 m²).
- Se deberán ubicar 200 toldos (1000 m²).
- Se deberán colocar 300 sombrillas (1500 m²).
- Se puede habilitar el bar, con lo cual se habilitará (100 m²).

Con esta configuración no quedan m² sin utilizar del balneario y la facturación es la máxima posible, con un valor de **\$1.565.000.000** en toda la temporada.

3. PROBLEMA 3

3.1. Enunciado y Supuestos

A Paolo Casanova siempre le gustó mucho viajar. Así que en 1994, mientras estaba cursando el CBC, se le ocurrió un algoritmo (ver anexo) que le daba la distancia mínima entre dos ciudades. El programa funcionaba bien y puso muy contento a Paolo, hasta que tiempo después se enteró que alguien más ya había desarrollado el algoritmo varios años antes.

El programa está escrito en **GW Basic**. Las fuentes se adjuntan en el anexo al presente enunciado y en archivo .bas. El intérprete y documentación del lenguaje se pueden encontrar en PC-BASIC.

3.2. Análisis del Algoritmo

La implementación del algoritmo brindado por la consigna está subido en el archivo `ejercicio_3.bas`, el cual muestra una implementación de un programa interactivo para que el usuario pueda ingresar una determinada cantidad de vértices y aquellas aristas que desee establecer entre los ejes con sus respectivos pesos, resultando así en un **grafo no dirigido ponderado** el cual, al ejecutar el programa, se selecciona un vértice de salida y permite mostrar todos los vértices del grafo y el camino mínimo hacia cada uno de los vértices.

Analizando el algoritmo, hemos llegado a la conclusión de que es una implementación que se asemeja al **Algoritmo de Dijkstra**, el cual es un algoritmo utilizado para grafos no dirigidos y ponderados, sin pesos negativos. Su funcionamiento es el siguiente: toma un vértice como *vértice de salida* y, para un *vértice de llegada*, busca el siguiente vértice que haga mínimo el camino completo. Este algoritmo, en su implementación, se apoya en un **heap de mínimos** para mejorar la búsqueda del siguiente vértice para que nos devuelva la arista más pequeña conectada con el vértice de esa iteración.

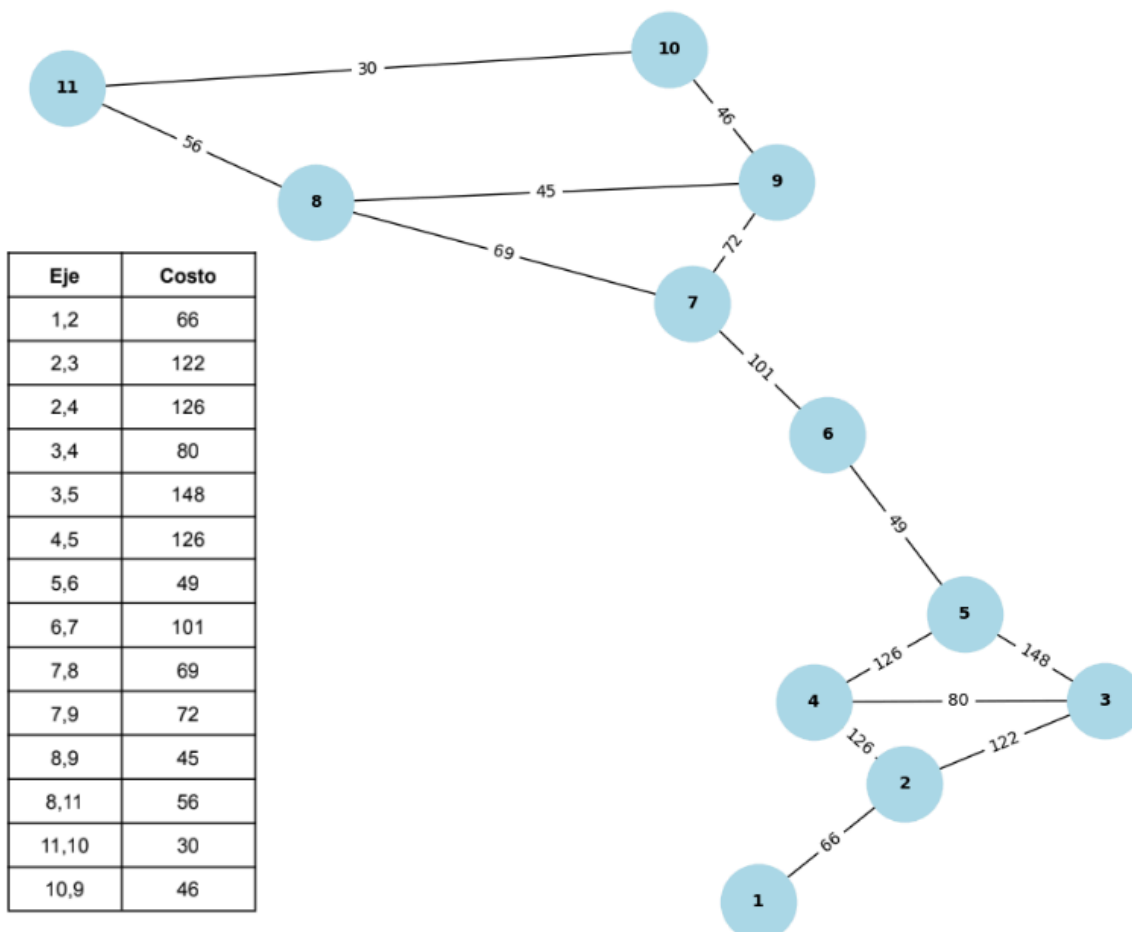
Cabe destacar, para la implementación de este **algoritmo Greedy** (cuya “regla sencilla” consiste en tomar el próximo vértice que tenga la arista más liviana y repetir la regla en cada iteración), el uso del heap de mínimos es un complemento para optimizar la búsqueda del próximo vértice y no forma parte de la implementación Greedy.

Teniendo estas consideraciones, y analizando la implementación de este algoritmo, puede identificarse numerosas diferencias entre la implementación propia del algoritmo y una implementación del *Algoritmo de Dijkstra*. Principalmente se destacan las diferencias entre las estructuras de datos utilizadas para el seguimiento del algoritmo y la detección de aristas. La implementación brindada por el enunciado utiliza una matriz de tamaño $N \times N$ y dos listas de tamaño N para poder realizar la ejecución correspondientemente (estas estructuras son explicadas más a detalle en la sección Estructuras de Datos). Debido a que no se utiliza un heap de mínimos, la búsqueda del vértice más óptimo no es del todo eficiente bajo determinados casos muy bordes, pero es posible asumir que las similitudes tanto de la

implementación de ambos algoritmos como los resultados que se obtienen son idénticos como para definir esta correlación.

3.3. Visualización con Datos de Prueba

Habiendo explicado el algoritmo brindado y el *Algoritmo de Dijkstra* el cual replica, de dispone en el enunciado una **Tabla de Prueba** con 11 vértices (ejes) en total, y 14 aristas que los conectan entre sí, conformando así el siguiente grafo:



Teniendo esta tabla de prueba, hemos decidido cargar los valores correspondientes tanto al algoritmo en *PC-Basic* como a una implementación del *Algoritmo de Dijkstra*, para poder comparar los resultados y no solo argumentar en las similitudes de las comparaciones en cada iteración que realizan ambos algoritmos, sino también para poder validar que los resultados son exactamente iguales. Cabe destacar que, al momento de cargar la *Tabla de Prueba* en *PC-Basic*, se ha tenido en cuenta múltiples consideraciones para que no se produzcan errores involuntarios e inesperados, los cuales pueden modificar la tabla de resultados (estos errores son señalados en la sección de **Errores Detectados**).

INGRESE EL VERTICE DE SALIDA ? 1				
SALIDA	LLEGADA	DISTANCIA		
1	1	0	Node 1 --> 1	-----> 0
1	2	66	Node 2 --> 1 2	-----> 66
1	3	188	Node 3 --> 1 2 3	-----> 188
1	4	192	Node 4 --> 1 2 4	-----> 192
1	5	318	Node 5 --> 1 2 4 5	-----> 318
1	6	367	Node 6 --> 1 2 4 5 6	-----> 367
1	7	468	Node 7 --> 1 2 4 5 6 7	-----> 468
1	8	537	Node 8 --> 1 2 4 5 6 7 8	-----> 537
1	9	540	Node 9 --> 1 2 4 5 6 7 9	-----> 540
1	10	586	Node 10 --> 1 2 4 5 6 7 9 10	--> 586
1	11	593	Node 11 --> 1 2 4 5 6 7 8 11	--> 593
OTRA VEZ? (SI/NO) ?				

En estas dos imágenes se puede visualizar los resultados de ambos algoritmos: en la izquierda se ve la tabla de resultados que imprime *PC-Basic* al finalizar la ejecución, mientras que la imagen de la derecha muestran los resultados obtenidos tras aplicar el *Algoritmo de Dijkstra* con los mismos datos. Como se pueden ver, los caminos óptimos para cada vértice tomando como vértice de salida el vértice 1 son idénticos en pesos para ambas implementaciones, brindando resultados idénticos.

3.4. Estructuras de Datos

En la implementación de este algoritmo, se utilizan numerosas **estructuras de datos** para poder cumplir con el objetivo del algoritmo. En cada caso, se mencionan las líneas del programa brindado por la cátedra (archivo `ejercicio_3.bas`) donde se encuentran presentes estas estructuras. Estas son:

- **COST** = Matriz de adyacencia de $N \times N$, donde el valor de cada casillero es el peso entre, por ejemplo, un vértice I y un vértice J (siendo I y J un número entre 1 y N). Si el valor de ese casillero I, J es **15.000 se considera como infinito o desconocido**.
 - Esta matriz es declarada en la línea 40, y se llena con los valores iniciales correspondientes en la línea 160 y 210. Luego, sus valores son utilizados para la inicialización de la lista **DIST** en la línea 260. Por último, sus valores son utilizados para el cálculo principal del algoritmo en la línea 1090.
 - Cabe destacar que se considera como un grafo dirigido el ingresado por el usuario, agregando siempre dos aristas en ambas direcciones cada vez que el usuario agrega una.
- **DIST** = Array que guarda la distancia del vértice seleccionado con todos los demás vértices, teniendo en cuenta que aquellos vértices que no son adyacentes están a una distancia “infinita”. Esta lista se actualiza a medida que se recorre el grafo y, una vez finalizada la ejecución del programa, imprime aquellos valores que se encontraron.

- Este array es declarado en la línea 50, y sus valores son inicializados mediante la matriz `COST` en la línea 260. Luego, sus valores son actualizados de forma definitiva en las líneas 1010 (asegurando que la distancia de un nodo con sí mismo es 0) y 1090 (con los valores reales, asumiendo que son vértices distintos). También es usado para el funcionamiento del algoritmo en la línea 1050. Por último, una vez finalizada la ejecución del mismo, se imprimen sus valores en la línea 320 y, en caso de que el usuario quiera volver a correr el programa con un nuevo vértice de origen, sus valores son limpiados a 0 en la línea 370.
- `SOL` = Corresponde a un array con flags booleanos que indican si el vértice en una determinada posición ya ha sido visitado (1) o si todavía no ha sido visitado (0).
 - Este array es inicializado en la línea 60 y sus valores son inicializados a 0 en la línea 270. En primer lugar, determina que el nodo de salida ya es visitado en la línea 1000. Luego, para el funcionamiento del algoritmo, sus valores son utilizados en la línea 1050 y actualizados en la línea 1070. Por último, una vez finalizada la ejecución del programa, en caso de que el usuario decida ejecutarlo de nuevo con un nuevo vértice de origen, sus valores son limpiados a 0 en la línea 370.

3.5. Complejidad Temporal y Espacial

Visualizando el código del ejercicio, se puede determinar la **Complejidad Temporal** analizando cada sección a detalle, considerando a $n = \text{cantidad de vértices del grafo}$, se observan las siguientes complejidades:

- La **Complejidad Temporal** es $O(n^2)$:
 - Inicializar la matriz de adyacencia `COST` cuesta $O(n^2)$.
 - Inicializar los arrays `DIST` y `SOL` cuesta $O(n)$.
 - Ingresar los datos por *stdin* cuesta $O(n)$.
 - La lógica principal tiene una complejidad de $O(n^2)$.
- La **Complejidad Espacial** es $O(n^2)$:
 - La matriz de adyacencia `COST` ocupa un espacio $O(n^2)$.
 - Los arrays ocupan $O(n)$.

3.6. Errores Detectados

Tras múltiples ejecuciones del programa, se han identificado **3 errores** que afectan al resultado final de la ejecución, los cuales pueden ser producidos tanto por la implementación del algoritmo en sí como por bugs menores los cuales terminan en comportamientos no deseados. Estos errores son los siguientes:

- En primera instancia, al momento de agregar aristas, si se posiciona el vértice de mayor tamaño delante del vértice de menor tamaño resulta en que, al tomar uno de aquellos

vértices de salida, esa arista agregada **no sea reconocida** al calcular los caminos mínimos. En este ejemplo, se puede ver como el camino desde el eje 2 a 3, y el camino desde 3 a 2, ambos tienen un peso de 20 implicando así un recorrido pasando por el eje 1 e ignorando por completo la arista 3, 2 ingresada con peso 10. Esto afecta en todos los casos donde esto se cumpla, incluido en la tabla de prueba brindada en el enunciado, donde las últimas dos aristas a ingresar son 11, 10 y 10, 9 respectivamente, lo que genera este problema.

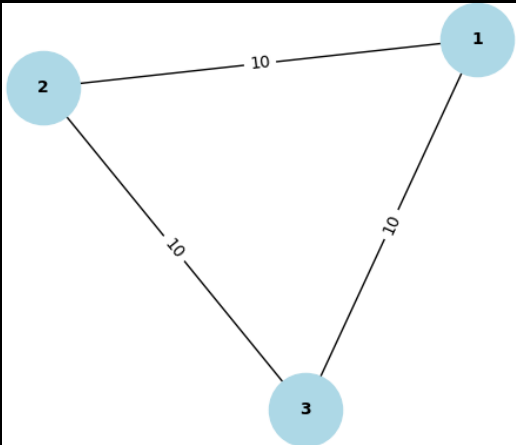
A continuación se muestra una captura de la ejecución del programa visualizando el ejemplo descrito:

```

INGRESE EL NUMERO DE VERTICES ? 3

INGRESE EL CUADRO DE COSTOS (INGRESE 0,0 PARA TERMINAR)

EL EJE ? 1,3
COSTO DEL EJE ? 10
EL EJE ? 1,2
COSTO DEL EJE ? 10
EL EJE ? 3,2
COSTO DEL EJE ? 10
EL EJE ? 0,0
INGRESE EL VERTICE DE SALIDA ? 2
SALIDA      LLEGADA      DISTANCIA
2           1           10
2           2           0
2           3           20
OTRA VEZ? (SI/NO) ? SI
INGRESE EL VERTICE DE SALIDA ? 3
SALIDA      LLEGADA      DISTANCIA
3           1           10
3           2           20
3           3           0
OTRA VEZ? (SI/NO) ? _
        
```

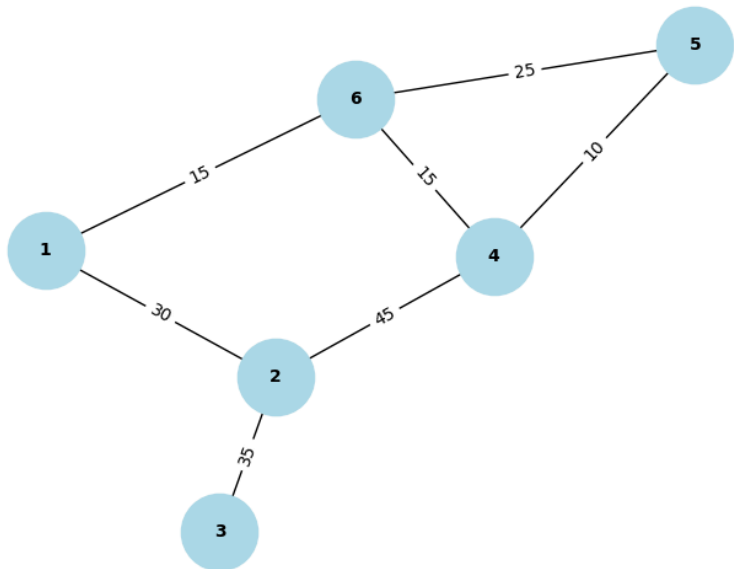


- En segundo lugar, otro error que hemos detectado que, bajo determinadas condiciones, afecta los resultados del algoritmo, es que tenga caminos largos entre dos nodos. En esos casos, estos caminos no se generan correctamente y el algoritmo considera como que tal camino no existe, por más de que sí lo haga y sea óptimo. Esto sucede justamente cuando se encuentran caminos con un largo mayor a 2 (es decir, que pasan por más de dos aristas), y se produce únicamente cuando el vértice de salida tiene un índice mayor que el vértice de llegada. Esto no sucede en cualquier otro caso. Por ejemplo, en la ejecución mostrada en la sección **Visualización con Datos de Prueba**, se toma como vértice de salida con índice 1 y todos los caminos son mostrados de forma correspondiente. Pero, si se quisiera ejecutar el algoritmo con un vértice de índice mayor o

igual a 5, aquellos ejes de llegada cuyos caminos tengan un largo mayor a 2 aristas no serán mostrados correctamente debido a que el algoritmo no los reconocería.

- El último error que hemos encontrado se produce cuando, el programa al ejecutar y marcar consecuentemente aquellos vértices como visitado con la lista SOL, puede terminar determinando caminos no óptimos desde el vértice de salida a uno determinado de llegada. Este error es más difícil de replicar (por ejemplo, no sucede en la ejecución mostrada en la sección **Visualización con Datos de Prueba**) pero, analizando aquellos casos donde si sucede, se puede deber a como está implementado el algoritmo y la detección de caminos que toma para definir un camino óptimo.

Estos dos últimos errores son mucho más complicados de replicar, pero en una ejecución simple pueden aparecer e interferir con el resultado que da la ejecución del programa. A continuación, se muestra un grafo (más simple que el visto anteriormente) donde se producen ambos errores en determinados casos:



En este grafo, en primer lugar, una vez cargado en *PC-Basic*, se puede observar claramente como existe un camino óptimo con vértice de salida 5 y vértice de llegada 3, con peso 90. Pero, debido al segundo error mencionado, como el índice del vértice de llegada es menor que el índice del vértice de salida y debido a que el camino óptimo pasa por más de dos aristas, éste no se ve reflejado en la tabla de resultados que se imprime al finalizar la ejecución. Esto se puede observar detalladamente en la tabla resultante de la ejecución del programa mostrada a continuación:

```
EL EJE ? 1,2
COSTO DEL EJE ? 30
EL EJE ? 1,6
COSTO DEL EJE ? 15
EL EJE ? 2,3
COSTO DEL EJE ? 35
EL EJE ? 2,4
COSTO DEL EJE ? 45
EL EJE ? 4,5
COSTO DEL EJE ? 10
EL EJE ? 4,6
COSTO DEL EJE ? 15
EL EJE ? 5,6
COSTO DEL EJE ? 25
EL EJE ? 0,0
INGRESE EL VERTICE DE SALIDA ? 5
SALIDA      LLEGADA      DISTANCIA
5           1           40
5           2           55
5           4           10
5           5           0
5           6           25
OTRA VEZ? (SI/NO) ? _
```

En esta otra imagen a la derecha se puede ver claramente cómo, si bien tomando el vértice con índice 5 como vértice de salida que anteriormente ha generado error con el vértice de llegada con índice 3, si tomamos el vértice 3 como salida y el vértice 5 como llegada, efectivamente muestra el camino mas optimo debido a que el índice del vértice de salida es menor que el índice del vértice de llegada.

INGRESE EL VERTICE DE SALIDA ? 3		
SALIDA	LLEGADA	DISTANCIA
3	1	65
3	2	35
3	3	0
3	4	80
3	5	90
3	6	95
OTRA VEZ? (SI/NO) ? SI		
INGRESE EL VERTICE DE SALIDA ? 6		
SALIDA	LLEGADA	DISTANCIA
6	1	15
6	2	45
6	3	80
6	4	15
6	5	25
6	6	0
OTRA VEZ? (SI/NO) ? _		

INGRESE EL VERTICE DE SALIDA ? 5		
SALIDA	LLEGADA	DISTANCIA
5	1	40
5	2	55
5	4	10
5	5	0
5	6	25
OTRA VEZ? (SI/NO) ? SI		
INGRESE EL VERTICE DE SALIDA ? 3		
SALIDA	LLEGADA	DISTANCIA
3	1	65
3	2	35
3	3	0
3	4	80
3	5	90
3	6	95
OTRA VEZ? (SI/NO) ? _		

Asimismo, en la imagen izquierda se puede identificar exactamente el último error mencionado anteriormente. Debido a la ejecución de la implementación de este algoritmo, tomando el vértice 3 como salida y el 6 como llegada, se devuelve el segundo camino más óptimo con peso 95. Sin embargo, si invertimos los vértices de salida y de llegada, se puede ver como el camino desde el vértice 6 hasta el 3 si da el resultado óptimo, con peso 80.

El segundo error, como fue mencionado anteriormente, se puede visualizar al ejecutar el programa y cargar el grafo entregado en el enunciado y explicado en la sección de **Visualización con Datos de Prueba**. En la siguiente imagen se puede ver específicamente como al tomar como vértice de salida el vértice de índice 7, se puede ver exactamente que aquellos vértices de llegada cuyos índices son menores y cuyos caminos pasan por más de 2 aristas, no son mostrados en la tabla final de la ejecución debido a que, por la implementación del algoritmo y el uso de las estructuras de datos, no se logran identificar aristas que conectan aquellos vértices faltantes, resultando como si quedaran a una distancia infinita o desconocida, como si fuese un **grafo no conexo**, cosa que no es verdad. También se puede

identificar tomando al vértice 5 como vértice de salida, sucediendo esto con el único vértice que cumple aquella caracterización: el vértice con índice 1.

INGRESE EL VERTICE DE SALIDA ? 7		
SALIDA	LLEGADA	DISTANCIA
7	5	150
7	6	101
7	7	0
7	8	69
7	9	72
7	10	118
7	11	125
OTRA VEZ? (SI/NO) ? SI		
INGRESE EL VERTICE DE SALIDA ? 5		
SALIDA	LLEGADA	DISTANCIA
5	2	252
5	3	148
5	4	126
5	5	0
5	6	49
5	7	150
5	8	219
5	9	222
5	10	268
5	11	275
OTRA VEZ? (SI/NO) ? _		

3.7. Posibles Mejoras

Teniendo en cuenta estos errores mencionados y la estructura del algoritmo, podemos sugerir como posible solución el uso de un **heap de mínimos** a beneficio para una mejor y más eficiente ejecución del algoritmo, sin tener errores correspondientes a la clasificación de aquellos vértices como “visitados”, y mejorando la búsqueda del siguiente vértice para que nos devuelva la arista más pequeña conectada con el vértice en cada iteración.

Asimismo, y para contribuir a la organización, legibilidad, mantenibilidad y desempeño del programa, hemos incluido en la entrega el archivo `ejercicio_3_refactor.py`, el cual divide en subrutinas aquellas partes del código diferentes entre sí para modularizar. Además, agregamos comentarios y una correcta indentación para poder visualizar aquellos ciclos y subrutinas que componen tanto la ejecución del programa (y su interacción con el usuario), como la ejecución del algoritmo.

REFERENCIAS

Documentación de Python. (2023). *Complejidad temporal*. [[TimeComplexity - Python Wiki](#)]

Tim Roughgarden. (2013). *Guide to Dynamic Programming*.

Software de Lindo Systems. *Lindo Systems Inc.* [[Home](#)]

Nicolas Gallagher. Jonathan Neal. (2018). *PC-BASIC*. [[PC-BASIC \(robhagemans.github.io\)](#)]

Muhammad Adeel Javaid. (2013). *Understanding Dijkstra's Algorithm*. CompTIA.