Taller: Aplicación de Algoritmos de Búsqueda

Nota sobre el desarrollo del taller

Para responder las preguntas correspondientes a este taller se tuvieron en cuenta varios aspectos, a continuación se enlistan:

1. Encontrará en el siguiente link el repositorio en GitHub de este taller: XYZ. En este encontrará los siguientes archivos:

```
Taller # 1 - Aplicación de Algoritmos de Búsqueda
 README.md               # Descripción de los ejercicios seleccionados y guía de uso
 taller_busqueda.ipynb   # Desarrollo y código de los ejercicios
 taller.pdf              # Documento con enunciados y reflexiones
```

2. Durante las preguntas seleccionadas encontrará que en todas se discutieron los siguientes aspectos:
   (a) Análisis del problema, que incluye el tipo de búsqueda que se solicita en el enunciado, el modelado de un grafo o múltiples grafos de ejemplos o una generalización del caso. Finalmente, se plantearán las cuestiones críticas planteadas en el enunciado.
   (b) Para cada algoritmo de búsqueda se utilizó la medida de distancia propuesta por el enunciado del algoritmo y se propone otra, que consideramos más apropiada para el acercamiento al problema.
   (c) Como medida de desempeño principal se tomo el tiempo de ejecución del algoritmo.
   (d) Se realiza una reflexión entre los distintos algoritmos de búsqueda para cada caso.

1 Ejercicio 1. Optimización de rutas rurales (logística)

Análisis del problema

Tipo de búsqueda: Búsqueda A*

¿Cómo funciona la Búsqueda A*?
El algoritmo A* es un método que permite encontrar la ruta más corta mediante una función de coste real del camino, $g(n)$, y una función heurística, $h(n)$, que proporciona una estimación del coste desde el nodo actual hasta el nodo objetivo, ofreciendo información sobre el camino restante. En particular, el coste total se calcula como:

$$f(n) = g(n) + h(n)$$

Donde:

- $g(n)$: coste real desde el inicio hasta el nodo actual.
- $h(n)$: coste estimado desde el nodo actual hasta el objetivo.

Modelado - Desarrollo:

En el cuaderno de programación, a partir del ejemplo del profesor, se solicita al usuario ingresar la estructura de un grafo en el siguiente formato: `[('A', 'B', 7), ('A', 'C', 9), ('B', 'D', 10), ('C', 'D', 2), ('C', 'E', 11), ('D', 'E', 1)]`. Posteriormente, esta lista se convierte en una tupla para poder declararla como grafo en el programa `networkx` (`nx`). Luego, se asignan coordenadas a los nodos con el fin de facilitar su representación gráfica.
A partir de una función, se define la métrica de distancia que utilizará el algoritmo; en este caso, se emplea la distancia euclidiana, que se define de la siguiente manera:

$$h(n) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Finalmente, mediante el paquete `networkx`, se calcula el camino más corto y su costo asociado.

Tiempo de ejecución (Métrica de desempeño):

Calidad de la heurística	Complejidad de tiempo aproximada	Explicación
Perfecta ($h(n) = h^*(n)$)	$O(d)$	Si la heurística siempre acierta el costo exacto al objetivo, A* sigue solo el camino óptimo, explorando casi sin ramas.
Admisible y consistente, pero débil	$O(b^d)$	Si la heurística aporta poca información, A* explora casi como un BFS uniforme.
Sin heurística ($h(n) = 0$)	$O(b^d)$	Se convierte en Uniform Cost Search, que es básicamente BFS ponderado.
Heurística efectiva	Entre $O(b^d)$ y $O(d)$	Cuanto más se acerque $h(n)$ al costo real, menos nodos se expanden y menor es el tiempo.

Table 1: Complejidad temporal de A* según la calidad de la heurística

Propuesta variación de distancia:

Optimización de A* con kilómetros reales y distancia geodésica
Para mejorar la precisión de un algoritmo A*, es fundamental diferenciar entre el **coste real del grafo** y la **heurística** utilizada. En el caso de un recorrido sobre coordenadas rurales, es recomendable asignar como *peso del grafo* ($g(n)$) los **kilómetros reales** de carretera entre nodos. Esto permite que el algoritmo optimice sobre distancias efectivas y no sobre aproximaciones lineales, generando rutas más realistas y acordes con las condiciones de la red vial.
Cuando el objetivo es optimizar el **tiempo de viaje**, el peso de cada arista puede definirse como el tiempo estimado, calculado a partir de la longitud real y la velocidad promedio o máxima asociada al tipo de vía:

$$w(e) = \frac{\text{longitud\_km}(e)}{\text{velocidad}(e)}$$

En cuanto a la *heurística* ($h(n)$), una alternativa robusta es emplear la **distancia geodésica** entre el nodo actual y el nodo objetivo. Este cálculo, basado en fórmulas como *Haversine* o *Vincenty*, considera la curvatura de la Tierra y proporciona una estimación mínima de la distancia que siempre será menor o igual a la distancia real por carretera. De esta forma, la heurística se mantiene **admisible** y **consistente**, garantizando que A* encuentre la solución óptima.
Si se optimiza la distancia, la heurística puede definirse como:

$$h_{\text{km}}(n) = \text{distancia\_geodésica\_km}(n, t)$$

Mientras que, si se optimiza el tiempo, es necesario dividir la distancia geodésica por la velocidad máxima permitida o estimada para la red:

$$h_{\text{tiempo}}(n) = \frac{\text{distancia\_geodésica\_km}(n, t)}{v_{\text{max}}}$$

Esto asegura que la estimación nunca sobrepase el coste real, condición necesaria para mantener la optimalidad de A*.
Finalmente, en contextos rurales, donde las velocidades pueden variar significativamente según el tipo de vía (pavimentada, destapada, terciaria), es recomendable ajustar tanto el peso del grafo como la heurística a partir de estas condiciones, manteniendo siempre las propiedades de **admisibilidad** y **consistencia** que hacen que A* funcione correctamente.

Cuestiones Críticas:

- **¿Cómo afecta la calidad de la heurística a la solución?¿Es pertinente usar distancia euclidiana?**

  La función heurística es la que nos proporciona un coste estimado desde el nodo actual hasta el nodo objetivo, dando información sobre el camino restante. Por lo que afecta sustancialmente qué tipo de distancia se escoja

para su cálculo y esto dependerá en gran parte del problema. En particular si nos enfocamos en el problema de logística rural, la distancia euclideana no nos serviría para esta aproximación, pues la distancia euclídea consiste en tomar dos puntos y trazar en línea recta entre estos dos.

- **¿Qué tan flexible sería su modelo ante la inclusión de nuevas rutas, restricciones o cierres viales?**

  En ese caso habría que notificar la estructura del grafo. A continuación se muestran las formas como se modifica la estructura de los grafos según el caso y un ejemplo.

- **¿Qué tanto escalaría su solución si el número de nodos de multiplica por 10?**

  Escalabilidad del Algoritmo A*

  Medir la escalabilidad de un algoritmo requiere analizar cómo se desempeñará a medida que el tamaño del insumo aumenta. Esto implica considerar alguna métrica específica, como el uso de memoria o el tiempo de procesamiento.

  En este caso particular, hablaremos de la **complejidad en tiempo** para llevar a cabo la tarea.

  La escalabilidad del algoritmo A* depende en gran medida de la definición de las funciones que representan el *coste real* desde el inicio hasta el nodo actual $g(n)$ y la *heurística* $h(n)$, que estima el costo desde el nodo actual hasta el objetivo. En particular, la función heurística es clave, pues es la que "poda" el espacio de búsqueda que explora el algoritmo.

  La complejidad del algoritmo A* puede crecer exponencialmente hasta:

  $$O(b^d)$$

  porque, en el peor de los casos, el algoritmo podría verse obligado a explorar un número de nodos que crece exponencialmente con la profundidad de la ruta óptima d (es decir, el número de nodos para llegar al destino). Esto ocurre si la función heurística es poco informativa, obligando al algoritmo a comportarse como una búsqueda a ciegas, similar a una búsqueda en amplitud (BFS).

  En este sentido, resulta importante resaltar el rol y la calidad de la heurística. El objetivo de la función heurística es guiar al algoritmo hacia el destino, permitiendo omitir la exploración de grandes porciones del espacio de búsqueda. A continuación, se describen distintos casos.

  Buena función heurística

  - **Heurística perfecta:** Si $h(n)$ es igual al costo real hasta el destino, el algoritmo solo explorará la ruta óptima, resultando en una complejidad lineal:

  $$O(d)$$

  - **Heurística admisible:** Una heurística es admisible si no sobrestima el costo real para llegar al destino. Una buena heurística admisible mantiene la complejidad cercana a la lineal.

  Mala función heurística

  Se considera que una función heurística es insuficiente cuando no proporciona al algoritmo una guía clara.

  - **Peor caso: explorar todos los nodos** En este escenario, la complejidad es exponencial:

  $$O(b^d)$$

  donde:

  - b: factor de ramificación, es decir, el número promedio de vecinos o conexiones por nodo. Un b alto implica más opciones por explorar en cada nivel.
  - d: profundidad de la ruta óptima. Un valor alto significa que la solución está más lejos del origen.

- **Heurística inadmisible:** Ocurre cuando la heurística sobreestima el costo real, lo que puede llevar al algoritmo a explorar muchas rutas subóptimas antes de encontrar la solución óptima.

- **Heurística inefectiva:** Cuando la función heurística no diferencia entre rutas prometedoras y malas, el algoritmo podría expandir casi todos los nodos de un nivel antes de pasar al siguiente, como en BFS.

- **Factor de ramificación alto (b):** Cuando el grafo tiene muchos vecinos por nodo, el crecimiento del espacio de búsqueda es muy rápido.

- **Profundidad de la solución grande:** Cuanto más lejos esté el destino, más nodos se deben explorar, y el tamaño de la frontera de búsqueda puede crecer exponencialmente con cada nivel.

Propuesta de variación de algoritmo de búsqueda:

El algoritmo **IDA*** (Iterative Deepening A*) es una variación del algoritmo A* que combina la búsqueda informada con la estrategia de profundización iterativa. A diferencia de A*, que mantiene en memoria una cola de prioridad con todos los nodos abiertos, IDA* realiza búsquedas en profundidad limitadas por un umbral sobre la función de evaluación:

$$f(n) = g(n) + h(n)$$

donde $g(n)$ es el coste acumulado y $h(n)$ la heurística admisible. Si no se encuentra la solución dentro del umbral actual, este se incrementa al menor valor de $f(n)$ que excedió el límite y el proceso se repite. Esta técnica reduce significativamente el consumo de memoria a $O(b \cdot d)$, manteniendo la optimalidad siempre que la heurística sea admisible y consistente. Por ello, IDA* resulta especialmente útil en problemas de gran escala, como la planificación de rutas en entornos rurales extensos, donde A* podría agotar los recursos de memoria disponibles.

2 Ejercicio 2. Red de metro (transporte)

Análisis del problema

Tipo de búsqueda: En este caso se solicitaba comparar dos algoritmos de búsqueda. El primero es el **Best First Search (BFS)**, el segundo se trata del algoritmo **Iterative Deepening Search (IDS)**.

¿Cómo funciona el Best First Search (BFS)?
El BFS Es un algoritmo de recorrido de grafos que explora un grafo o árbol nivel por nivel. Este algoritmo visita todos los vecinos inmediatos del nodo fuente antes de pasar al siguiente nivel de nodos. Es decir, todos los nodos de la misma profundidad se procesan antes de seguir al siguiente nivel de profundidad. Todos los posibles caminos se recorren. En el siguiente gráfico se muestra la forma de operación del algoritmo:

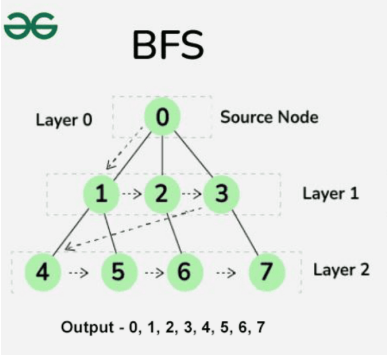

Figure 1: Operación del algoritmo BFS

1. Paso 1: Explorar todos los nodos en el primer nivel.
2. Paso 2: Moverse al siguiente nivel y explorar todos los nodos.
3. Paso 3: Repetir con el siguiente nivel.
4. Paso 4: Repetir con el último nivel.

¿Cómo funciona el Iterative Deepening Search (IDS)?
El algoritmo IDS (Iterative Deepening Search o Búsqueda en Profundidad Iterativa) combina las ventajas de la búsqueda en anchura (BFS) y la búsqueda en profundidad (DFS), ya que encuentra soluciones óptimas en grafos no ponderados como BFS, pero con un consumo de memoria reducido como DFS. Su funcionamiento consiste en realizar sucesivas búsquedas en profundidad limitadas, empezando con un límite de profundidad igual a cero y aumentando dicho límite en cada iteración hasta encontrar la meta. En cada paso, se exploran todos los nodos hasta el nivel permitido y, si no se halla la solución, se incrementa el límite y se repite el proceso. Este enfoque garantiza que la primera vez que se alcanza la meta sea a la menor profundidad posible, asegurando así la optimalidad, aunque implica reexplorar nodos varias veces, lo que puede aumentar el tiempo de ejecución en grafos muy grandes o densos. En el siguiente gráfico se presenta la operación del algoritmo IDS:

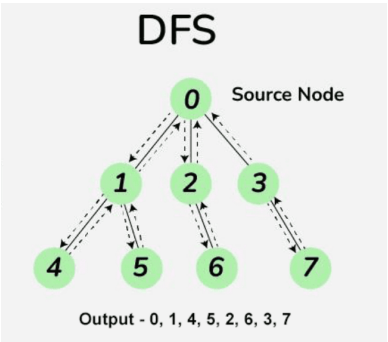

Figure 2: Operación del algoritmo IDS

1. Paso 1: Seleccionar un nodo inicial y marcarlo como visitado.
2. Paso 2: Visitar el primer vecino no visitado del nodo actual.
3. Paso 3: Repetir el paso anterior hasta que no haya más vecinos no visitados.
4. Paso 4: Retroceder al nodo anterior y buscar otros vecinos no visitados.
5. Paso 5: Repetir este proceso hasta que todos los nodos alcanzables hayan sido visitados.

Modelado - Desarrollo:

En el cuaderno de programación se encuentra el grafo en particular del que se tratará este punto. Este consiste en 10 nodos, que corresponden a una línea de metro. En este punto lo que se quiere encontrar es el camino más corto entre la estación A a la J. A continuación, se presenta una representación del nodo

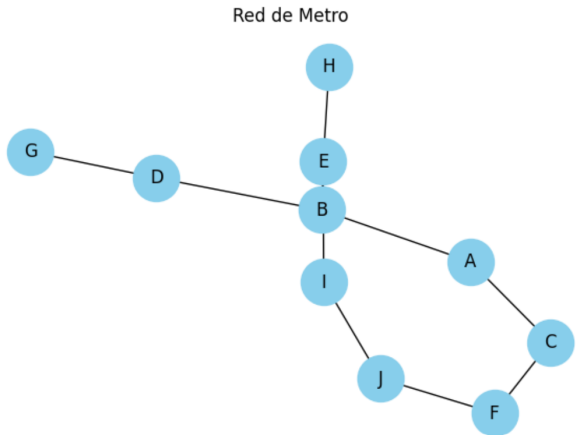

Figure 3: Representación grafo de la red del Metro

Para esto comparamos el algoritmo BFS e IDS, a continuación se explica qué se hizo en cada caso:

1. **BFS:** Se definió una función que tiene como insumo tres valores, el primero es el grafo, el segundo es el nodo de inicio y el tercero es el nodo objetivo. En esta misma función se definen dos variables importantes, la primera, "visited", es un diccionario que acumula los nodos ya visitados y la segunda es una variable diccionario, "queue" de aquellos caminos hacia los nodos que hacen falta visitar, pues en este algoritmo se recorren todas las rutas posibles.

   Posterior a esto, se realiza un bucle de exploración de recorrido, que se ejecuta mientras la estructura de datos de frontera, en este caso "path", no esté vacia. Si el algoritmo encuentra elementos, comienza por el primero camino que encuentra. El nodo en el que se fija, es el último que encuentra. Posterior a esto, con un condicional, si el nodo actual es el objetivo, termina y devuelve el camino por el que llego a él. Si no es el objetivo, explora los vecinos del nodo actual. Para cada vecino que no ha sido visitado, crea un nuevo camino añadiendo el vecino al camino actual y lo añade a la cola.

2. **IDS:** La función principal del algoritmo IDS es ir aumentando progresivamente el límite de profundidad para encontrar la solución más cercana. Por lo que el algoritmo comienza definiendo una función con tres insumos (el nodo de inicio, el nodo de destino, los vecinos adyacentes y un límite de profundidad máxima). En este bucle se establece un límite de profundidad, mientras se realiza un recorrido de profundidad sin traspasarlo. Inicialmente si el nodo inicial es el objetivo, este devuelve el camino por el que lo encontró. Pero mientras haya nodos en la fila, mira el último elemento del stack. Si ya se ha llegado a la profundidad máxima permitida se quita ese nodo de la fila, y volvemos a la siguiente iteración de la profundidad. Si hay uno nodo vecino se explora el nodo vecino, pero si no hay más vecinos significa que se ha terminado de explorar el nodo y lo quitamos de la pila y del camino. Finalmente, si la fila está vacía y no encontramos la meta, se devuelve que no hubo éxito.

Propuesta variación de distancia:

En este caso dado que nos encontramos frente a un grafo no ponderado, la distancia que se contempla por defecto es el número de nodos a recorrer. Como propuesta de variación de distancia se podría pensar en el tiempo de recorrido de cada ruta o el número de kilometros entre cada estación.
Si se implementara esta métrica este grafo tendría ponderaciones y se podría sustituir la medida de distancia utilizada. Para proponer una variación de la distancia en particular, se plantea una estructura de grafo de la siguiente manera:

$$\begin{aligned} \text{G3} = \{&('A','B',5),\ ('A','C',7),\ ('B','D',3),\ ('B','E',4),\\ &('C','F',6),\ ('D','G',8),\ ('E','H',2),\ ('E','I',5),\\ &('F','J',9),\ ('I','J',4)\} \end{aligned}$$

Aquí cada peso representa el número de minutos que se demora el Metro recorriendo ese camino.

*Nota: En el cuaderno de programación encontrará la implementación de los dos algoritmos de búsqueda a partir de esta variación de la métrica de distancia.

Tiempo de ejecución (Métrica de desempeño):

Algoritmo	**Complejidad de tiempo (Big-O)**	**Observaciones de rendimiento**	**Escenarios donde es mejor**
BFS (Breadth-First Search)	$O(b^d)$	Explora todos los nodos nivel por nivel. El tiempo crece exponencialmente con la profundidad d. Encuentra el camino más corto en número de aristas de forma directa.	Cuando la solución está cerca de la raíz (poca profundidad) o cuando se necesita el camino más corto garantizado rápidamente.
IDS (Iterative Deepening Search)	$O(b^d)$	Misma complejidad asintótica que BFS, pero con un factor constante mayor por repeticiones. Los nodos cercanos a la raíz se visitan varias veces, lo que introduce un overhead de tiempo (∼10–40% más que BFS en promedio).	Cuando el espacio de estados es muy grande y no cabe en memoria (IDS consume $O(b \cdot d)$ memoria).

Table 2: Comparación general del tiempo de ejecución entre BFS e IDS

Cuestiones Críticas:

- **¿Qué pasa si el sistema crece a 100 nodos? ¿Cuál es el impacto en la complejidad temporal?**
  En términos de notación asintótica, el coste de tiempo del algoritmo IDS y BFS se analiza con dos parámetros:
  - b: factor de ramificación, es decir, el número promedio de vecinos o conexiones por nodo. Un b alto implica más opciones por explorar en cada nivel.
  - d: profundidad de la ruta óptima. Un valor alto significa que la solución está más lejos del origen.

  Para el caso del algoritmo **BFS** este explora todos los nodos de cada nivel hasta llegar a d, así que el número total de nodos generados crece exponencialmente con b, de la siguiente manera:

  $$O(b^d)$$

  En el caso del algoritmo **IDS** en el peor de los casos, este algoritmo crecería a la misma razón. Sin embargo, si el nodo destino se encuentra antes de recorrer todos los nodos el costo temporal disminuiría. Por otro lado, es importante mencionar que en términos de memoria este tiene un sobrecosto adicional, que se evidencia como una constante, de la siguiente manera

  $$O(b * d)$$

  En este caso solo guarda la ruta actual y los hijos inmediatos de la fila.

- **¿Puede automatizarse la elección del mejor algoritmo según el tamaño de la red?**
  Si se podría realizar una implementación de una toma de decisiones a partir del tamaño del nodo, sin embargo, esta sería una aproximación muy vaga y general, pues no hay nada que nos indique sobre la profundidad de la solución d ni el factor de ramificación b. Resumiendolo en una regla práctica:

  - Si $n \leq$ presupuesto de memoria en nodos $\Rightarrow$ usa BFS
  - Si $n >$ presupuesto $\Rightarrow$ usa IDS

  El presupuesto de memoria en nodos es básicamente un número máximo estimado de nodos que la memoria puede mantener al mismo tiempo sin quedarse sin RAM. A continuación, la estimación de esta:

$$\text{presupuesto\_memoria\_nodos} \approx \frac{\text{RAM disponible para el algoritmo}}{\text{Memoria por nodo}}$$

Propuesta de variación de algoritmo de búsqueda:

Los algoritmos de búsqueda realizados anteriormente funcionan para encontrar la ruta más óptima, estos no son los más eficientes en términos temporales y de memoria. Por lo que se podría utilizar un algoritmo A* para mejorar la eficiencia, pues este contempla qué tan prometedor puede ser un nodo. Por lo que puede ser mejor en rutas largas, si la función heurística es buena.

3 Ejercicio 3. Filogenia (biología)

Análisis del problema

Tipo de búsqueda:

Un grafo acíclico dirigido (a menudo denominado grafo acíclico dirigido o DAG) es un grafo que: Tiene aristas dirigidas (las flechas van de un nodo a otro en una dirección específica). No contiene ciclos (no hay forma de comenzar en un nodo y seguir una secuencia de aristas que finalmente vuelva al nodo de partida).

¿Cómo funciona la Búsqueda de Profundidad Limitada?

Un grafo acíclico dirigido (DAG) es un grafo formado por nodos conectados por aristas que tienen una dirección y no forman ciclos. Esto significa que no se puede empezar en un nodo y volver a él siguiendo las flechas. Los DAG son útiles para representar procesos con un orden claro, como programaciones de tareas, flujos de trabajo de datos o historiales de versiones. Su naturaleza acíclica permite operaciones como la clasificación topológica, el recorrido de grafos y la búsqueda de rutas sin riesgo de bucles infinitos.

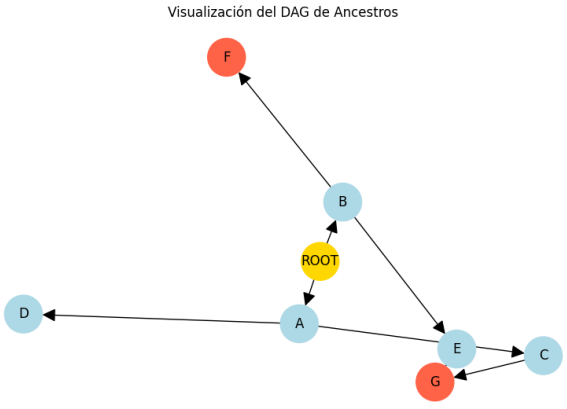

Figure 4: Enter Caption

Modelado - Desarrollo:

Se implementaron scripts de Python para ilustrar la construcción de un grafo acíclico dirigido (DAG), acompañados de ejemplos prácticos. Para incorporar la automatización, mejorar el nivel de complejidad y examinar aspectos

técnicos adicionales de los DAG, en particular el antepasado común más reciente (MRCA), se desarrolló una aplicación interactiva. Esta herramienta permite al usuario seleccionar dos o más nodos de forma dinámica, visualizar sus relaciones y resaltar la ruta MRCA, lo que facilita una comprensión más completa de otros conceptos.

Tiempo de ejecución (Medida de desempeño)

Ejecución secuencial Si las tareas se ejecutan una tras otra, el tiempo total es simplemente:

$$T_{\text{total}} = \sum_{n \in N} T(n)$$

donde $T(n)$ es el tiempo de ejecución del nodo n y N es el conjunto de nodos.

Cuestiones Críticas:

- **¿Cómo afecta la elección de profundidad límite?**

  Cuando se utiliza la búsqueda limitada en profundidad (DLS) en un (DAG) para encontrar el antepasado común más reciente (MRCA) de dos nodos, la elección del límite de profundidad es fundamental. El límite de profundidad define hasta dónde se puede explorar la búsqueda desde un nodo inicial. Si este límite se establece demasiado bajo, es posible que la búsqueda no recorra el gráfico con la profundidad suficiente para llegar a los antepasados comunes reales de los nodos en cuestión.

  Por ejemplo, un DAG simple en el que el nodo 1 es el antepasado raíz de los nodos 2 y 3, y el nodo 4 es descendiente tanto del 2 como del 3. El MRCA de los nodos 2 y 4 es el nodo 1. Sin embargo, si el límite de profundidad se establece en solo 1, la búsqueda desde el nodo 4 solo encontrará sus padres inmediatos (los nodos 2 y 3) y no el nodo 1. Como resultado, el algoritmo informaría incorrectamente de que no hay un antepasado común, simplemente porque no ha buscado lo suficientemente lejos.

  En términos prácticos, esto significa que un límite de profundidad demasiado restrictivo reduce la potencia de su algoritmo de búsqueda. Puede pasar por alto relaciones reales que existen dentro del gráfico, lo que debilita el propósito mismo de la búsqueda. Por otro lado, establecer un límite de profundidad demasiado alto puede aumentar el tiempo de cálculo, pero garantiza la exhaustividad: encontrará todos los antepasados posibles y, por lo tanto, el verdadero MRCA. Es pues, el punto más importante a la hora de considerar la profundidad como un hiperparámetro y encontrar el punto óptimo donde no se consideren todas las relaciones posibles -lo cual es muy costoso computacionalmente-, pero sin limitar la misma profundidad y no encontrar el MRCA esperado.

- **¿Podría automatizarse este análisis para múltiples pares?**

  Por supuesto, este análisis se puede ampliar más allá de los pares de nodos a grupos de cualquier tamaño, como tríos, cuartetos o conjuntos más grandes. La idea central es identificar el antepasado común más reciente (MRCA) compartido por todos los nodos del grupo. Para lograrlo, el algoritmo calcula el conjunto de antepasados de cada nodo y luego determina la intersección de estos conjuntos, lo que da como resultado los nodos que son antepasados comunes a todos los miembros del grupo. Entre estos, el MRCA se define como el más cercano (es decir, con la profundidad o distancia combinada mínima) a los nodos en cuestión.

  Este enfoque puede automatizarse para cualquier número de grupos de nodos; sin embargo, a medida que aumenta el tamaño de los grupos y la complejidad del DAG, también aumenta el costo computacional. No obstante, el principio sigue siendo el mismo: el MRCA de cualquier grupo es el antepasado que comparten todos los nodos del grupo y al que se llega con la longitud total más corta de la ruta desde todos los miembros del grupo. Esta generalización se utiliza ampliamente en filogenética, genealogía y análisis de redes.

- Este análisis se puede extender fácilmente a grupos de más de dos nodos, como tríos o cuartetos. Por ejemplo, supongamos que tenemos tres nodos: v_1, v_2 y v_3. Primero se calcula el conjunto de ancestros para cada uno:

  $$A(v_1) = \{a, b, c\}, \quad A(v_2) = \{b, c, d\}, \quad A(v_3) = \{b, c, e\}$$

  Luego se busca qué ancestros tienen en común, es decir, se calcula la intersección:

  $$A^* = A(v_1) \cap A(v_2) \cap A(v_3) = \{b, c\}$$

  Finalmente, el MRCA del grupo será el ancestro que esté más cerca de los tres nodos, es decir, el que se pueda alcanzar en menos pasos desde cada uno. Si b está más cerca que c, entonces b sería el MRCA.

Este procedimiento se puede automatizar para cualquier cantidad de nodos. Eso sí, cuanto mayor sea el grupo y más complejo sea el grafo, más tiempo puede tardar el análisis.

En este caso, el algoritmo se modificó para ir más allá de la simple identificación de pares de nodos. Se implementó de manera interactiva, permitiendo al usuario seleccionar los nodos específicos para los que desea encontrar el antepasado común más reciente (MRCA) que pueden ser más de 3 si así el usuario lo requiere, hasta 40 nodos. Una vez realizada la selección, el algoritmo resalta el MRCA en amarillo, muestra la ruta óptima calculada y presenta la profundidad optimizada, lo que proporciona una visualización más clara y eficiente de los resultados.

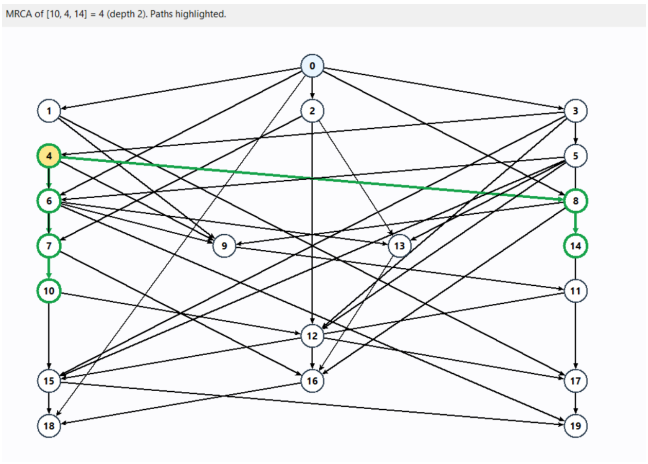

Figure 5: MRCA para mútiples nodos

- **¿Qué tan escalable sería este sistema con árboles filogenéticos reales con cientos de especies?**

  Como se mencionó anteriormente, este tipo de método es muy adecuado para detectar ancestros comunes. Sin embargo, la complejidad de los problemas del mundo real hace que estos cálculos requieran más recursos. Aun así, el sistema se puede escalar y aplicar a árboles filogenéticos con cientos de especies. No obstante, la eficiencia depende del tamaño y la complejidad del grafo. A medida que aumenta el número de nodos y conexiones, también aumenta el tiempo necesario para encontrar todos los conjuntos de ancestros y su intersección.

  En la práctica, existen potentes herramientas de software como **RAxML** (**stamatakis2014raxml**), que se utiliza ampliamente en bioinformática y está optimizada para analizar de manera eficiente árboles con cientos o incluso miles de especies. La mayoría de los programas filogenéticos utilizan algoritmos avanzados y estructuras de datos eficientes para manejar grandes conjuntos de datos en plazos razonables. No obstante, en el caso de grafos extremadamente grandes o densos, puede ser necesario utilizar técnicas de optimización adicionales para mejorar el rendimiento.

Ejercicio 4. Decisiones de inversión (economía)

Análisis del problema

Tipo de búsqueda: Una estrategia de búsqueda *voraz* (greedy search), cuando se aplica a un árbol de decisiones de inversión, representa un caso particular de una búsqueda heurística del tipo "primero el mejor" (*best-first search*). El principio operativo fundamental de este algoritmo es la optimización local en cada etapa del proceso de decisión.

¿Cómo funciona el Greedy Search?

La naturaleza "voraz" del algoritmo se manifiesta en su criterio de selección: elige la opción que es localmente óptima, es decir, la que promete el mayor beneficio inmediato, sin realizar una evaluación prospectiva de las consecuencias a largo plazo de dicha elección.

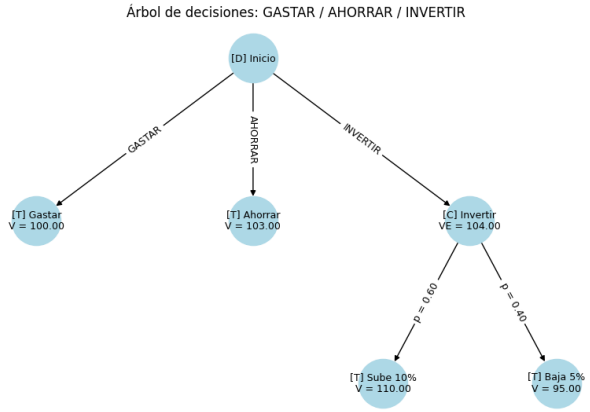

Figure 6: Estructura

– **Heurística de Decisión:** La función de utilidad esperada, $U(a)$, sirve como la función heurística que guía la búsqueda. La simplicidad de esta heurística garantiza una baja complejidad computacional.

$$a^* = \arg\max_{a \in A_n} U(a)$$

– **Ausencia de Garantía de Optimalidad Global:** La principal limitación de esta estrategia es que una secuencia de decisiones localmente óptimas no garantiza la consecución de la trayectoria que maximizaría la utilidad total acumulada. Es decir, el óptimo local no implica necesariamente un óptimo global. Un camino descartado inicialmente por una baja utilidad inmediata podría haber conducido a oportunidades de mayor valor en etapas posteriores.

En conclusión, la búsqueda voraz en este contexto es un método algorítmico eficiente y directo que se fundamenta en una heurística de maximización inmediata, pero que carece de la presciencia necesaria para asegurar un resultado globalmente óptimo en el espacio de todas las políticas de inversión posibles.

Un árbol de decisiones de inversión representa nodos de acción económica (gasto, ahorro, inversión). Utilice búsqueda greedy sobre un árbol ponderado de utilidad esperada.

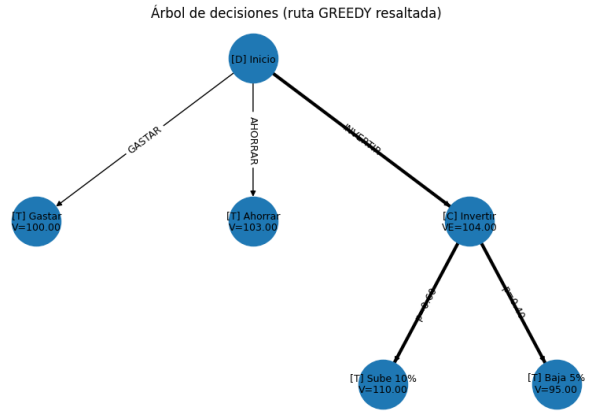

Figure 7: Greedy Search

Modelado - Desarrollo:

El sistema modela un escenario de finanzas personales como un árbol de decisión jerárquico con tres opciones principales: *gastar*, *ahorrar* o *invertir*. La rama Ahorrar contiene cinco bancos, a cada uno de los cuales se le asigna una tasa de interés y una comisión fija extraídas de rangos predefinidos. Los rendimientos aquí son deterministas, calculados como el crecimiento compuesto durante el horizonte de inversión menos la comisión, con una variabilidad insignificante. La rama Invertir se divide en Criptomonedas y Acciones, cada una de las cuales contiene cinco activos seleccionados de sus respectivos fondos. El rendimiento de los activos se modela utilizando un proceso lognormal: el rendimiento logarítmico durante el horizonte elegido tiene una

media μT y una varianza $\sigma^2 T$, lo que permite el cálculo en forma cerrada del valor esperado y la desviación estándar de cada activo.

El algoritmo de toma de decisiones sigue una estrategia codiciosa, clasificando las opciones según una puntuación ajustada al riesgo definida como:

$$\text{Score} = E[\text{Payoff}] - \lambda \cdot \text{Std}(\text{Payoff})$$

Aquí, λ actúa como parámetro de aversión al riesgo. El modelo evalúa la opción *Gastar*, el mejor banco en *Ahorrar* y el activo con mejor rendimiento en *Invertir*, seleccionando la puntuación más alta sin realizar una búsqueda más profunda. Esto hace que el marco sea sencillo, pero sensible al capital (C_0), el horizonte temporal (T), la preferencia de riesgo (λ) y el muestreo aleatorio de los parámetros de los activos.

La interfaz interactiva se implementa con controles deslizantes y botones de `matplotlib` para ajustar C_0, T, λ y la semilla aleatoria. El botón *Resample* (Remuestrear) regenera los conjuntos de activos, mientras que un registro de ejecución documenta las decisiones recientes. Se utiliza `NetworkX` para visualizar el árbol de decisión con etiquetas de nodos y bordes resaltados para la ruta elegida. El código es modular, separando la evaluación de la visualización y la lógica de la interfaz de usuario, lo que permite tanto pruebas independientes como la exploración interactiva.

Propuesta variación de distancia

lugar de usar únicamente la métrica actual del algoritmo codicioso —basada en el valor esperado menos un término de penalización por riesgo—, se podrían incorporar otras formas de medir la "distancia" de cada opción respecto a un objetivo deseado. Por ejemplo, esta distancia podría ser la desviación frente a una meta de rentabilidad, la cercanía a un punto ideal de rendimiento y riesgo, o la diferencia respecto a la frontera eficiente. También podría medirse en términos probabilísticos, como la probabilidad de no alcanzar un objetivo, o mediante un valor en riesgo condicional (CVaR), lo que permitiría evaluar no solo el promedio, sino también los escenarios desfavorables. Cada una de estas alternativas cambiaría el criterio de selección y, por tanto, la ruta elegida en el árbol de decisión.

Los sliders de la interfaz podrían adaptarse para controlar los parámetros asociados a cada nueva métrica de distancia. Por ejemplo, ajustar el nivel de riesgo aceptable, la meta de capital final o la tolerancia a pérdidas. Incluso podría añadirse un control para cambiar de forma interactiva la métrica activa, de manera que el usuario pueda observar cómo varían las decisiones al modificar la forma de medir la "mejor" opción. Esto haría posible una exploración más amplia y comparativa de estrategias dentro del mismo modelo.

Tiempo de ejecución (Métrica de desempeño)

En un **greedy search**, el tiempo de ejecución crece aproximadamente de forma lineal con la profundidad d del árbol, ya que en cada nivel el algoritmo evalúa las b opciones posibles y elige únicamente la que presenta la mejor heurística. Si t_{eval} es el costo de evaluar un nodo, se tiene:

$$T_{\text{greedy}} \approx O(b \cdot d \cdot t_{\text{eval}}) \quad \text{y, para } b \text{ pequeño,} \quad T_{\text{greedy}} \approx O(d)$$

Este comportamiento contrasta con algoritmos como BFS, cuyo costo puede crecer de forma exponencial $O(b^d)$, lo que hace que la búsqueda codiciosa sea más eficiente en tiempo para problemas con gran profundidad y bajo factor de ramificación.

Además, para la métrica de desempeño, se utilizó una fórmula más amplia donde se condieraran los hiperparámetros funcionales dentro del código de python en el archivo ejecutable llamado $\text{punto}_4 interactivo.pynb.Acontinuaci$

$$T_{\text{ejec}} \approx b \cdot d \cdot t_{\text{eval}}(C_0, T, \lambda, \text{seed})$$

- b: factor de ramificación.
- d: profundidad del árbol.
- t_{eval}: tiempo de evaluación por nodo, dependiente de los parámetros controlados por los *sliders*:
  - C_0: capital inicial.
  - T: horizonte temporal.
  - λ: aversión al riesgo.
  - seed: semilla para la generación aleatoria de activos.

Cuestiones críticas

- **¿Es válida la heurística en contextos volátiles? ¿Cómo se valida o ajusta esa métrica?**

  En entornos volátiles, la heurística del valor esperado (X) es insuficiente por sí sola, pues ignora probabilidades inestables, eventos extremos y la aversión al riesgo. Tiende a ser una métrica excesivamente optimista.

Validación de la Heurística

Para verificar su aplicabilidad, se debe recurrir a:

- **Backtesting:** Evaluación con datos históricos.
- **Simulación Monte Carlo:** Modelado de múltiples escenarios estocásticos.
- **Análisis de Sensibilidad:** Medición de la robustez del resultado ante cambios en los parámetros.

Ajustes para Incorporar el Riesgo

Para que la heurística sea más robusta, se debe ajustar mediante:

- **Función de Utilidad $U(x)$:** Reemplazar el valor monetario x por una función que modele la aversión al riesgo (e.g., $U(x) = \log(x)$ o $U(x) = \sqrt{x}$).
- **Métricas Riesgo-Retorno:** Utilizar ratios que ponderen el rendimiento por su volatilidad, como el Ratio de Sharpe o Sortino.
- **Penalización por Varianza:** Modificar la métrica para castigar la incertidumbre. Por ejemplo:

  $$\text{Valor Ajustado} = X - k \cdot X$$

  donde k es el coeficiente de aversión al riesgo.
- **Métricas de Escenarios Adversos:** Incorporar el impacto de los peores casos, utilizando medidas como el *Conditional Value at Risk* (CVaR).

¿Cómo podría incorporar adaptabilidad o aprendizaje sobre decisiones anteriores?

Incorporación de aprendizaje

La adaptabilidad del árbol de decisiones se logra mediante la modificación de probabilidades, utilidades o estructura a partir de la experiencia:

- **Actualización bayesiana:** recalcular probabilidades tras cada resultado observado.
- **Aprendizaje por refuerzo:** usar algoritmos como Q-Learning para ajustar valores esperados según recompensas.
- **Pesos dinámicos:** ponderar decisiones según desempeño histórico.
- **Memoria de decisiones:** registrar acciones y resultados para detectar patrones.
- **Simulación adaptativa:** reestimar utilidades con Monte Carlo y datos recientes.
- **Meta-decisiones:** elegir heurísticas distintas según métricas recientes o nivel de riesgo.

Flexibilidad ante cambios económicos

La verdadera flexibilidad frente a cambios en el entorno económico depende de cómo se implementen ciertos mecanismos de actualización y aprendizaje.

Actualmente, el modelo es flexible gracias a:

- Simulación Monte Carlo.
- Penalización por riesgo.
- Aprendizaje histórico.
- Score dinámico.

Sin embargo, presenta limitaciones como probabilidades y volatilidades estáticas, estructura rígida y falta de variables macroeconómicas.

Modificaciones recomendadas

- Actualizar probabilidades y volatilidades periódicamente.
- Incorporar variables macroeconómicas (inflación, tasas de interés).
- Permitir expansión dinámica del árbol.
- Implementar meta-decisiones estratégicas.

Ejemplo contextual

Si la inflación aumenta:

- Disminuir utilidades de opciones como cuentas de ahorro y CDT.
- Penalizar liquidez excesiva.
- Favorecer activos reales (inversiones, criptomonedas).

En conclusión, un árbol de decisiones con actualización continua, sensibilidad al entorno y aprendizaje se convierte en una herramienta robusta para entornos inciertos para la toma de decisiones, en este caso de inversión de capital. Sin embargo, y como en todas las herramientas, se debe conocer el contexto, considerar las distintas variaciones de los elementos de los que están sujetos el comportamiento del modelo y la susceptibilidad al cambio de los mismos.