**Diálogos y Prompts con IA:**

Consultar a Inteligencias Artificiales resulto de suma importancia para obtener explicaciones claras sobre las características de los grafos y el funcionamiento lógico y practico de los distintos algoritmos de consulta vistos en clase. Además, de resultar muy útiles para realizar comparaciones entre enfoques, detección de errores y brindar ejemplos prácticos adaptados al conocimiento propio del alumno y el material brindado por el profesor de la catedra.

1. **“Podrías explicarme cómo funciona la heurística en el algoritmo de A\*, para saber cómo el algoritmo entiende qué nodos seguir y cuáles no. Menciona a qué hace referencia con la distancia Manhattan.”**

**-Para que sirvió:** Nos permitió esclarecer las dudas respecto al algoritmo de A\*, ya que no poseíamos conocimientos previos sobre la función heurística y como el algoritmo hacia uso de este para encontrar un camino. Además, no comprendíamos el funcionamiento del método Manhattan y su relación con la heurística.

**-Lo aprendido:** La función heurística es una estimación de qué tan cerca se está del objetivo. Teniendo la idea central de decidir qué nodo explorar primero según un puntaje f(n). Se tiene la función f(n) = g(n) + h(n), siendo g(n) es el costo real desde el nodo inicial hasta el nodo n (es decir, lo que ya se recorrió), y h(n) la heurística (estimación del costo para llegar desde n hasta el objetivo). Básicamente, la heurística es la “intuición” del algoritmo sobre qué camino parece más prometedor. Si h(n) es pequeña, significa que el nodo n está estimado cerca del objetivo. Por otro lado, la distancia Manhattan es un tipo de heurística para tableros cuadrados (por ejemplo, un laberinto, un mapa en grilla). Los movimientos se limitan en líneas rectas horizontales o verticales, no en diagonal. Teniendo dos puntos, la función se calcula haciendo h(n) = |x1 – x2| + |y1 – y2|. Siendo |x1 – x2| el número de casillas que hay que moverse horizontalmente. Y |y1 – y2| es el número de casillas que hay que moverse verticalmente.

Para saber el camino que debe tomar A\*, primero mantiene una cola de prioridad de nodos a explorar, basada en f(n) = g(n) + h(n). Siempre explora el nodo con menor f(n). Luego, calcula g(n) a medida que avanza y actualiza h(n) según la heurística elegida (por ejemplo, la Manhattan). De esta forma, evita nodos que aumentarían demasiado f(n) y continúa hasta que el nodo objetivo se extrae de la cola, garantizando el camino más corto si la heurística es admisible.

1. **“Cómo logra el algoritmo de Bellman-Ford, detectar ciclos negativos y al mismo tiempo poder trabajar con pesos negativos. Explícame detalladamente el concepto de relajación de aristas y cómo se vincula con el algoritmo.”**

**-Para que sirvió:** Permitió entender en profundidad la funcionalidad del algoritmo de Bellman – Ford, ya sea con su vinculación con los ciclos negativos y con los pesos negativos. Al mismo tiempo, ver qué diferencias posee si se lo compara con el algoritmo de Dijkstra.

**-Lo aprendido:** La relajación de aristas es la operación para propagar mejoras en las estimaciones de distancia desde la fuente. La relajación sólo propaga la información de que existe una mejor ruta. Repetir relajaciones en todo el grafo deja que las mejores distancias fluyan desde la fuente hacia todos los nodos. Bellman – Ford usa la relajación como bloque de construcción y aplica relajación sobre todas las aristas repetidamente. Repite la operación de relajar todas las aristas V-1 veces, siendo V la cantidad de vértices. Si después de esas V-1 iteraciones aún existe alguna arista que puede relajarse, entonces hay un ciclo de peso negativo alcanzable desde la fuente. La razón de que sean V-1 iteraciones, radica en que cualquier camino simple (sin vértices repetidos) entre dos vértices contiene a lo sumo V-1 aristas. Si no hay ciclos negativos, la mejor distancia a cualquier nodo se alcanza con algún camino simple, por lo tanto, relajando todas las aristas V-1 veces cualquier mejora habrá llegado a su destino.

Bellman – Ford funciona con pesos negativos porque no asume que una vez que se fija un nodo como “final” su distancia es definitiva. No sella distancias, continúa relajado y actualizado hasta converger (o detectar un ciclo negativo). Si se lo compara con Dijkstra, este último asume que cuando se extrae el nodo con menor “dist” de la cola de prioridad, esa distancia es la distancia definitiva mínima a ese nodo. Esto no es cierto si existen aristas negativas, puede ocurrir que más adelante aparezca una ruta más barata que llega a ese nodo pasando por otro con coste reducido por un peso negativo.

1. **“Podrías hacerme una comparación detallada entre los siguientes algoritmos: BFS, Dijkstra, Floyd-Warshall, Bellman-Ford, A\* y Prim/Kruskal. Ten en cuenta la complejidad utilizada por cada uno. El tipo de grafo con el que trabajan, ya sea dirigido, no dirigido, ponderado, no ponderado, utilizando pesos negativos.”**

**- Para que sirvió:** Permitió ver de una forma clara y concisa las diferencias entre los algoritmos. De esta forma es más fácil entender las características de cada uno.

**- Lo aprendido:** BFS encuentra las distancias mínimas en grafos no ponderados (o con todos los pesos iguales a 1) y sirve para recorridos por capas o niveles. Funciona en grafos ya sean dirigidos o no dirigidos, siempre y cuando no sean ponderados y su complejidad es O(V + E) usando lista de adyacencia. BFS sirve como base para otros algoritmos.

Dijkstra encuentra caminos mínimos desde un origen a todos los demás vértices. Funciona en grafos dirigidos o no dirigidos, de los cuales deben ser ponderados, sin embargo, no soporta pesos negativos. Su complejidad puede ser O(V2) con matriz y O((V + E) log V) con lista y cola de prioridad.

Bellman – Ford encuentra caminos mínimos desde un origen a todos los vértices, siendo capaz de detectar ciclos negativos. Funciona en grafos dirigidos o no dirigidos, de los cuales deben ser ponderados. Su complejidad es O(V \* E).

Floyd – Warshall: Encuentra caminos mínimos entre todos los pares de vértices. Funciona para grafos dirigidos o no dirigidos, de los cuales deben ser ponderados. También, soporta pesos negativos, pero no detecta ciclos negativos. Su complejidad es O(V3).

A\* encuentra el camino mínimo desde un origen hasta un destino específico. Funciona en grafos dirigidos o no dirigidos, de los cuales deben ser ponderados con pesos mayores o iguales a 0. No funciona con pesos negativos, dado que la heurística pierde su validez. Su complejidad es O(E log V).

Prim encuentra el árbol recubridor mínimo. Funciona para grafos no dirigidos, con ponderaciones mayores o iguales a 0. Su complejidad es O(V2) con matriz y O(E log V) con heap.

Kruskal encuentra un árbol recubridor mínimo (MST). Funciona para grafos no dirigidos y ponderados con pesos mayores o iguales a 0. Su complejidad es O(E log E).

1. **¿Cuál es el principal uso de la librería SFML 2.6.x en C++; Es posible implementarlo en algoritmos de consulta de grafos?**

**-Para que sirvió:** Conocer el propósito de la librería SFML 2.6.x nos permite entender en que contextos podemos aplicarla dentro de nuestros proyectos en C++. Aprendimos a diferenciar entre una librería de propósito general para gráficos y multimedia, y los algoritmos teóricos de grafos que normalmente implementamos en consola. Esto nos da la capacidad de pensar como unir ambos aspectos: la teoría algorítmica con la representación visual, lo que facilita el análisis, la depuración y la enseñanza de estructuras y procesos en grafos.

**-Lo aprendido:** La librería SFML 2.6.x en C++ se utiliza principalmente para el desarrollo de aplicaciones gráficas y videojuegos en 2D, ofreciendo herramientas para manejar gráficos, sonido, ventanas y eventos de usuario. En el contexto de los algoritmos de consulta de grafos (como BFS, Dijkstra, A\*), se puede usar SFML para visualizar gráficamente el grafo y los pasos del algoritmo, mostrando como se exploran los nodos, se relajan aristas o se construye un camino mínimo. Esto no solo hace más intuitiva la comprensión de los algoritmos, sino que también permite crear simuladores o entornos interactivos para experimentar con diferentes configuraciones de grafos.

1. **Necesito que me expliques como se implementa el sistema de caminos bloqueados (muros) en el algoritmo de A\* y como puedo programarlo en C++.**

**-Para que sirvió:** Implementar caminos bloqueados (también llamados muros u obstáculos) en A\* es fundamental cuando queremos simular entornos reales donde no todas las rutas están disponibles. Nos es especialmente útil para su implementación en aplicaciones como navegación de robots, videojuegos, planificación urbana o simulaciones logísticas. Además, nos enseña a incorporar restricciones espaciales para el diseño de algoritmos adaptativos, que responden a condiciones dinámicas del entorno.

**-Lo aprendido:**

* 'vector<vector<int>> grid;' — crea la matriz del mapa; cada celda contiene 0 = libre, 1 = muro (o un coste grande si quieres permitir paso caro).
* 'setWall(x,y) -> grid[y][x] = WALL;' — marcar una celda como muro impide usarla en la búsqueda (y es la forma más simple de “colocar” un obstáculo).
* 'struct Node {int x,y; float g,h,f; Node\* parent;};' — guarda posición, coste desde start (g), heurística (h), f=g+h y el padre para reconstruir el camino.
* 'priority\_queue ordered by f (open set)' — cola que siempre extrae el nodo con menor f; aquí se elige el siguiente a expandir.
* 'closed[y][x] = true' — matriz para no reprocesar nodos ya expandidos.
* 'dx, dy arrays' — define movimientos (por ejemplo 4 direcciones: {0,1},{1,0},{0,-1},{-1,0} o 8 si permites diagonales).
* En el bucle principal: extraer current de open; si current == goal -> reconstruir ruta siguiendo parent hasta start.
* Para cada vecino: calcular nx = current.x + dx[i], ny = current.y + dy[i]; comprobar límites (0<=nx<w, 0<=ny<h); luego comprobar muro: if (grid[ny][nx] == WALL) continue; — esta línea es la que bloquea el paso y evita añadirlo al open set.
* Calcular tentative\_g = current.g + move\_cost (1 ó √2 para diagonales); si tentative\_g < neighbor.g entonces actualizar neighbor.g = tentative\_g, neighbor.h = heuristic(nx,ny,goal), neighbor.f = g+h, neighbor.parent = current y push/update en la cola.
* Continuación/optimización práctica: usar una estructura para actualizar claves en la heap (o aceptar duplicados y filtrar con closed), elegir heurística admisible (Manhattan para 4-dir, Euclid/Octile para diagonales) para mantener optimalidad, y si los muros cambian en tiempo real considerar invalidar nodos afectados o usar algoritmos incrementales (D\* Lite) en vez de recomputar todo.