Práctica 1 – Búsqueda A* en Mapas de Terreno (Convocatoria C4, Curso 2024/2025)

Introducción

En esta memoria se documenta de forma rigurosa el desarrollo de la Práctica 1 de **Sistemas Inteligentes** (Universidad de Alicante, conv. C4 2024/25). El objetivo de la práctica es implementar y analizar el algoritmo de búsqueda **A** *y sus variantes en un entorno de mapas de terreno, comparando diferentes heurísticas y extensiones (como Weighted A* **y** Aε*****) para evaluar su comportamiento. Se trabaja sobre un escenario de laberinto/grid 2D donde un agente (un conejo) debe encontrar un camino hasta un objetivo (zanahoria) esquivando obstáculos y terrenos de distinto "costo" (energético).

En primer lugar, explicaremos brevemente el funcionamiento del algoritmo A *clásico y presentaremos un ejemplo detallado de su ejecución en un problema pequeño, incluyendo cálculo de los valores* g*,* h *y* f *para ilustrar la traza de búsqueda. A continuación, se describirá la implementación realizada, respondiendo específicamente a cómo se han manejado las estructuras de datos clave: la lista frontera (abierta), la lista interior (cerrada), la lista focal (usada por A*ε) y la representación de los estados del problema.

Seguidamente, se comparan varias **heurísticas** (Euclídea, Manhattan y Chebyshev, entre otras) evaluando su admisibilidad e influencia en el rendimiento (nodos generados/expandidos), apoyándonos en gráficas comparativas. Después, se presentan experimentos con la variante **A*ε** (A *con subóptimo controlado por* ε*): se prueba el algoritmo con distintos valores de* ε *y se analiza el impacto en el coste de la solución y el número de nodos explorados, así como el consumo en "calorías" (energía) del camino obtenido. Se incluye también una comparativa directa entre A* y Aε*****, mostrando mediante gráficas las diferencias en coste del camino y calorías consumidas para casos de estudio concretos.

Más adelante, investigamos la **técnica de ajuste de pesos** en la heurística (algoritmo Weighted A*), ejecutando pruebas con diferentes valores del parámetro de peso* w *y discutiendo cómo afecta al equilibrio entre optimalidad del camino y eficiencia computacional. También se describen brevemente varios mapas de prueba****** empleados, cada uno ilustrando escenarios particulares: uno con caminos muy largos, otro sin solución posible, y otro con múltiples rutas alternativas.

Por último, se proporcionan **instrucciones claras para la ejecución** del programa desarrollado, de modo que cualquier usuario (aun sin conocimientos técnicos profundos) pueda ejecutar y probar la aplicación paso a paso. Se concluye con una bibliografía que referencia el material del temario de la asignatura y otras fuentes utilizadas.

Funcionamiento del Algoritmo A* – Explicación y Ejemplo Paso a Paso

El algoritmo **A** *es un método de búsqueda heurística informada que encuentra el camino de costo mínimo desde un estado inicial hasta un estado objetivo en un grafo (o grid, en nuestro caso), combinando de forma*

equilibrada la exploración por costo real acumulado y por estimación heurística restante. A mantiene dos funciones de costo para cada estado (casilla del mapa):

- **g(n)**: el coste acumulado desde el origen hasta el estado *n* (distancia recorrida real).
- **h(n)**: la heurística que estima el costo desde *n* hasta el objetivo (distancia restante estimada, típicamente admisible).

A *evalúa cada estado mediante f(n) = g(n) + h(n), que representa una estimación del costo total del camino pasando por* n*. En cada iteración expande (explora) el estado con menor* f *de la lista frontera (pendientes por explorar). Si la heurística* h *es admisible (nunca sobrestima el costo real restante) y consistente, A* está garantizado a encontrar una solución óptima (de mínimo costo) si existe. En nuestro caso usamos heurísticas de distancia que son admisibles (como se discutirá más adelante).

Ejemplo ilustrativo: Consideremos un mapa pequeño de 5x5 celdas con un estado inicial *S* (fila 1, col 1) y objetivo *G* (fila 3, col 3), más algunos obstáculos. El algoritmo A *(heurística euclídea) explorará los estados como se muestra en la Figura 1 (se indica en cada celda el orden de generación/expansión y sus valores* f=g+h* calculados):

- *Inicialización:* El estado inicial *S* comienza con **g=0** (costo recorrido = 0) y se calcula su heurística **h = dist_euclid(S,G)** (distancia euclídea hasta *G*). Por ejemplo, si *S=(1,1)* y *G=(3,3)*, entonces $h \approx 2.828$ ($\sqrt{((3-1)^2+(3-1)^2)}$). Así, **f = g + h = 0 + 2.828 ≈ 2.83**. Este estado inicial se inserta en la lista frontera (prioridad más baja *f*).

- *Primer paso:* Se extrae *S* de la frontera (es el único estado con f≈2.83). Se generan sus estados vecinos (celdas adyacentes no bloqueadas). Supongamos que desde *S* podemos mover a dos vecinos libres: *N1* y *N2*. Para cada vecino, se calcula:
  – g(N) = g(S) + costo_mov(S→N). (Aquí el costo de movimiento horizontal/vertical es 1, diagonal es 1.5 en nuestro entorno).
  – h(N) = dist_euclid(N,G).
  – f(N) = g(N) + h(N).
  Imaginemos que *N1* está en posición (1,2) (movimiento horizontal, costo 1) y *N2* en (2,2) (movimiento diagonal, costo 1.5). Entonces:

- Para *N1*: g=1; h≈2.236 (euclídea a (3,3)); f≈3.236.

- Para *N2*: g=1.5; h≈1.414; f≈2.914.
  Se insertan *N1* (f≈3.236) y *N2* (f≈2.914) en la lista frontera. *S* pasa a la lista interior (explorados).

- *Segundo paso:* De la frontera, el estado con menor f es *N2* (f≈2.914). Se expande *N2*: se calculan sus vecinos (no explorados previamente). Supongamos que *N2* genera un nuevo vecino *N3* que resulta ser el objetivo *G*. Para *N3=G*:
  – g(G) = g(N2) + costo(N2→G). Si *N2=(2,2)* y *G=(3,3)*, este movimiento diagonal cuesta 1.5, por lo que g(G)=1.5+1.5=3.0.
  – h(G) = 0 (el objetivo, distancia restante 0).
  – f(G) = 3.0 + 0 = 3.0.
  G se inserta en la frontera.

- *Tercer paso:* Ahora en la frontera están *N1* (f≈3.236) y *G* (f=3.0). El estado con menor f es *G*. Al extraerlo, vemos que es el estado objetivo, por lo que el algoritmo termina. Se reconstruye el **camino solución** retrocediendo los padres desde *G* hasta *S*. En nuestro ejemplo, el camino encontrado podría ser *S* → *N2* → *G*, con un coste total real **g=3.0** (que coincide con el óptimo) y

número de pasos 2 (diagonales). Todas las demás celdas examinadas (como *N1*) quedan como exploradas pero descartadas en la solución final.

En la Figura 1 se observa la traza descrita: los números indican el orden de exploración de las celdas, y en cada celda explorada se podría anotar su valor *f* correspondiente. Sólo las celdas marcadas con línea continua amarilla forman finalmente parte del camino óptimo hallado (de *S* a *G*), mientras que otras exploradas quedan como intentos descartados.

En resumen, A *expande iterativamente los estados más prometedores según* f. *Esto le permite encontrar rutas óptimas evitando explorar exhaustivamente todas las posibilidades (como haría una búsqueda no informada). En nuestro pequeño ejemplo, A* examinó apenas unos pocos estados antes de localizar la meta. En problemas mayores, la eficiencia de A *dependerá en gran medida de la calidad de la heurística h(n)** elegida, como veremos a continuación.

Detalles de Implementación de la Solución

A continuación se describen las decisiones de implementación tomadas para las estructuras principales del algoritmo:

Representación de los estados

Los **estados** del problema corresponden a las posiciones en el mapa (coordenadas de celda en la cuadrícula). En la implementación, cada estado se maneja mediante una instancia de la clase Nodo=, que almacena la información necesaria para A*: la posición (fila, columna), un puntero al padre (el nodo desde el cual se llegó a este estado, para reconstruir la ruta), los costos acumulados g, heurísticos h y total f, así como las calorías** acumuladas (energía consumida) hasta ese estado.

Cada celda del mapa tiene asociado un tipo de terreno que influye en las calorías gastadas al atravesarla (pero *no* en el costo g de movimiento, que en nuestra implementación depende sólo de la distancia: 1 para movimientos ortogonales y 1.5 para diagonales). Los códigos de terreno son: 0 suelo normal (hierba), 4 agua, 5 roca, y 1 muro (obstáculo infranqueable). Al leer el mapa de un fichero de texto, estos códigos se cargan en una matriz. Un estado se considera identificado únicamente por su coordenada en el mapa; por ello, para marcar visitados se usa la posición como clave (no el objeto nodo completo).

Lista frontera (open list)

La **lista frontera** es la estructura que almacena los estados generados pero pendientes de expandir (también llamada lista abierta). Se ha implementado como una **cola de prioridad** ordenada por el valor *f* de los nodos. En Python se utilizó heapq (un min-heap) para gestionar la prioridad: cada vez que se inserta un nodo nuevo, este se posiciona según su *f*. Al momento de expandir, se extrae siempre el nodo con menor *f* del heap (operación O(log n)).

Adicionalmente, se mantiene un **diccionario** (`in_frontier`) que mapea coordenadas de estado -> objeto Nodo correspondiente en la frontera. Esto nos permite, al generar un vecino que ya esté en la frontera, **actualizar sus costos** si encontramos una ruta más barata (*g* menor) en lugar de insertar un duplicado. Tras actualizar un nodo existente, se reordena el heap (`heapq.heapify`) para restaurar la

propiedad de prioridad. De esta forma, evitamos múltiples copias del mismo estado en la frontera y aseguramos siempre tomar la versión de menor coste.

Lista interior (closed list)

La **lista interior** (o lista cerrada) es el conjunto de estados ya explorados/expandidos, que no deben volver a explorarse. Se implementó simplemente como un **conjunto (set)** de coordenadas. Cada vez que un nodo es extraído de la frontera para expandirse, su coordenada se añade al conjunto cerrado. Si durante la generación de vecinos encontramos uno que ya esté en la lista cerrada, lo ignoramos (en la implementación, esto está implícito porque no volvemos a insertar estados ya cerrados; si apareciera un intento de reinsertar, se detecta ya sea porque está en el diccionario de frontera o se comprueba el set de cerrados antes de expandir). Esto previene ciclos y re-expansiones innecesarias.

Nota: En nuestra implementación no se realiza la reapertura de nodos cerrados aunque se encuentre posteriormente un camino mejor hacia ellos (esta situación no ocurre si la heurística es consistente; de lo contrario, se podría omitir algún óptimo, pero nuestras heurísticas principales son consistentes/admisibles).

Lista focal (para A*ε)

La **lista focal** es una estructura adicional utilizada en la variante *Aε. Recordemos que Aε* (también llamada *A* con suboptimalidad limitada) mantiene dos criterios: un criterio principal similar a A *(valor f) y un criterio secundario (en nuestro caso, minimizar calorías). En la implementación, seguimos el enfoque de crear, en cada iteración de Aε*, una lista focal con los nodos de la frontera cuyo *f* cumple **f ≤ (1+ε)*f_min**, siendo *f_min* el menor valor f en toda la frontera. Es decir, se toma el subconjunto de nodos casi tan prometedores como el mejor (dentro de un factor *(1+ε)*). Entre esos nodos de la focal, el algoritmo selecciona para expandir aquel con **menor consumo de calorías** (atributo `calorias`).

En cuanto a implementación, cuando se va a elegir el siguiente nodo a expandir en A*ε, se filtran los contenidos del heap frontera para construir la lista focal (simplemente con una list comprehension) y luego se aplica* `min()` *sobre esa lista focal con la clave de calorías. Ese nodo seleccionado se retira de la frontera. (El resto de la lógica de generación de vecinos es similar a A* normal, con la diferencia de que *f* siempre se calcula con *g+h* sin peso). La lista focal como tal no se mantiene persistente entre iteraciones, sino que se construye sobre la marcha en cada selección.

Cálculo de heurísticas y costo de movimientos

Para completar, destacamos cómo se calculan los valores de costo y heurística en el código:

- La función de **costo de movimiento** entre celdas adyacentes (`calcular_coste`) devuelve un par: (costo_distancia, calorías_terreno). El costo de distancia es 1 para movimientos verticales/horizontales y 1.5 para diagonales. El costo en calorías depende del terreno de la celda destino: mover a una celda de hierba (.) consume 2 calorías, a agua (~) 4 calorías, a roca (*) 6 calorías, mientras que si la celda es muro (#) el movimiento es imposible (se retorna infinito para marcar bloqueo). Estos valores modelan que ciertos terrenos son más "costosos" energéticamente, aunque no penalizan la distancia.

- La **heurística h(n)** por defecto se tomó como la **distancia euclídea** en el grid desde *n* hasta la meta: `h = sqrt((Δx)^2 + (Δy)^2)`. Esta heurística es admisible ya que en un movimiento en grid 8-direcciones la distancia real mínima nunca excederá la euclídea (aunque en nuestro

caso el movimiento diagonal cuesta 1.5 en lugar de $\sqrt{2} \approx 1.414$, la euclídea sigue siendo $\leq$ costo real, por lo que es admisible y de hecho consistente). También se implementaron otras heurísticas para comparación, como se detalla en la siguiente sección.

Con estas estructuras, el algoritmo A* quedó implementado siguiendo el pseudocódigo estándar: se inicializa la frontera con el nodo inicial; se itera extrayendo el nodo con menor f, comprobando si es objetivo y si no, expandiendo sus vecinos (calculando g/h/f y manejando pertenencia a abierta/cerrada como explicado). La función devuelve la ruta reconstruida (usando los punteros padre) y la suma de calorías del camino encontrado, o indica fallo si no encuentra solución.

Comparativa de Heurísticas: Euclídea vs Manhattan vs Chebyshev

La elección de la función heurística *h(n)* es crucial en el rendimiento de A*. Se han evaluado tres heurísticas de distancia habituales en grids para nuestro problema:

- **Distancia euclídea**: *h_euclid(n) = sqrt((x_n - x_goal)^2 + (y_n - y_goal)^2)*. Es la heurística base que usamos en A*; intuitivamente mide la línea recta hasta la meta. Es admisible (nunca sobrestima la distancia real con movimientos diagonales de costo 1.5) y consistente, por lo que A* con esta *h* encuentra caminos óptimos.

- **Distancia Manhattan**: *h_manh(n) = |x_n - x_goal| + |y_n - y_goal|*. Suma las diferencias en horizontal y vertical. En un grid con movimientos diagonales permitidos, la Manhattan tiende a **sobreestimar** el costo real (por ejemplo, en línea recta diagonal, Manhattan da un valor mayor que el camino diagonal realmente existente). Por ello **no es admisible** en nuestro caso – podría guiar A* por un camino subóptimo. Sin embargo, probamos su efecto en la exploración por curiosidad. En general, Manhattan tiende a expandir menos nodos (al ser más agresiva) pero puede perder optimalidad.

- **Distancia Chebyshev**: *h_cheb(n) = max(|Δx|, |Δy|)*. Esta heurística equivale al número de movimientos diagonales necesarios si se pudiera ir en línea recta (considerando que en movimientos 8-direccionales el mínimo número de pasos viene dado por el mayor de los desplazamientos horizontal/vertical). Es admisible en nuestro escenario (de hecho para costo diagonal =1.5 sigue siendo $\leq$ costo real). Suele ser menos informativa que la euclídea (da valores menores o iguales a h_euclid porque ignora componentes adicionales de distancia).

Para comparar estas heurísticas, se ejecutó el algoritmo A* *en escenarios de prueba representativos midiendo el número de nodos generados/expandidos* hasta encontrar la solución (y verificando si la solución es óptima). La Figura 2 muestra, por ejemplo, la cantidad de nodos explorados con cada heurística en un cierto mapa de prueba:

En nuestros experimentos, **Manhattan** resultó explorar la menor cantidad de nodos (debido a que sobrestima y dirige más “golosa” la búsqueda hacia la meta). Por ejemplo, en un mapa con meta en diagonal respecto al inicio, Manhattan expandió un ~30% menos de nodos que Euclídea. Sin embargo, cabe destacar que Manhattan **no garantiza caminos óptimos**: en escenarios con obstáculos, al sobreestimar podría ignorar rutas viables más largas y terminar el algoritmo con una solución subóptima. En las pruebas realizadas, Manhattan casualmente halló rutas de igual longitud que las

óptimas en distancia (no empeoró el coste en nuestros casos simples), pero esto es una coincidencia; teóricamente podría fallar en optimalidad.

La heurística **Euclídea** expandió un poco más de nodos que Manhattan (por ser más conservadora) pero siempre obtuvo la ruta óptima. Es un buen compromiso entre admisibilidad y precisión: al ser una estimación más cercana al costo real que Chebyshev, redujo significativamente el esfuerzo de búsqueda frente a heurísticas más débiles, sin dejar de garantizar óptimo.

Por su parte, la **Chebyshev** (admisible) resultó ser la menos informativa de las tres en mapas abiertos: exploró ligeramente más nodos que la euclídea (alrededor de un 5–10% más en nuestras pruebas). Esto era esperable ya que h_cheb(n) $\leq$ h_euclid(n) para todos los estados, lo que hace que A *con Chebyshev tenga valores* f* menos diferenciados y por tanto abra más caminos en paralelo antes de descartar alternativas. No obstante, Chebyshev también garantiza optimalidad y podría ser útil en grids con movimiento diagonal unitario (coste = 1) donde coincidiría con la distancia real máxima en un eje.

En resumen, la heurística euclídea fue elegida como **heurística principal** en nuestra implementación por ser admisible y ofrecer un buen rendimiento (menor número de nodos que Chebyshev, sin los riesgos de Manhattan). Esta decisión asegura que el algoritmo A* encuentre siempre la ruta óptima en términos de distancia recorrida.

(Para referencia, todos los métodos heurísticos mencionados fueron implementados en el código y podían alternarse; sin embargo, en la versión final entregada se dejó la euclídea como predeterminada por ser "la más adecuada" como indicaba el enunciado.)

Búsqueda A*ε – Resultados con Distintos Valores de ε**

La variante **A*ε** introduce una relajación en el criterio óptimo de A*: permite encontrar soluciones no necesariamente mínimas, acotando su suboptimalidad por un factor* (1+ε)*, a cambio de priorizar algún otro criterio secundario (en nuestro caso, minimizar calorías consumidas). Esto es útil cuando existen rutas alternativas con costo ligeramente mayor pero que pueden ser preferibles bajo otro criterio (energético, tiempo, etc.).

Implementación recapitulada: A*ε mantiene la misma lista frontera ordenada por* f *que A*, pero en cada iteración selecciona para expandir no necesariamente el nodo de menor f global, sino el de menor calorías entre aquellos cuya f $\leq$ (1+ε)*f_min. Con ε=0 se recupera el comportamiento de A* (estrictamente menor f), y a medida que ε crece se permite explorar nodos con f más grandes (peores en costo) si ofrecen ventajas en calorías.

Se realizaron pruebas de A*ε en mapas con terrenos de distinto coste para observar cómo varía la calidad de la solución (coste en pasos) y el consumo de calorías, según* ε*. En general, los resultados muestran lo siguiente:

- Con **ε muy pequeño** (p.ej. 0.1), la solución hallada por A*ε suele coincidir con la óptima de A* en cuanto a distancia. El algoritmo apenas se desvía del camino de costo mínimo, por lo que las calorías consumidas también serán similares a las de la ruta óptima (que no necesariamente minimiza calorías). El número de nodos explorados también permanece cercano al de A *puro, pues la condición f $\leq$ 1.1*f_min restringe fuertemente la focal. En nuestras pruebas, para ε=0.1 no se apreciaron diferencias significativas: A*ε(0.1) exploró prácticamente los mismos nodos que A* y encontró rutas con igual coste.

- Al **incrementar ε**, A*ε comienza a considerar rutas alternativas de costo un poco mayor. Esto puede traducirse en encontrar caminos que evitan terrenos costosos. Por ejemplo, en un mapa donde la ruta óptima en distancia cruzaba una zona de rocas (alto consumo), con un valor moderado ε=0.3–0.5 el algoritmo encontró una ruta un ~20–30% más larga en pasos pero que bordea la zona, reduciendo las calorías totales considerablemente. En un caso de prueba, A* (ε=0) dio un camino de 7 pasos atravesando rocas con 20 calorías de gasto, mientras que A*ε(0.5) halló un camino de 9 pasos alrededor de las rocas con sólo 14 calorías consumidas. Esa mejora energética se logra a costa de 2 pasos extra (suboptimalidad ~28% en longitud) y también de mayor trabajo computacional (A*ε tuvo que explorar más nodos opcionales antes de decidirse por dicha ruta alternativa).

- Con **ε grandes** (≥1.0), la búsqueda se vuelve mucho más permisiva en costo. En el extremo ε=1.0 (equivale a permitir hasta el doble del costo mínimo), la lista focal abarca prácticamente cualquier nodo con f hasta 2*f_min, por lo que el algoritmo puede desviarse fuertemente de la ruta más corta si ello produce beneficios en el segundo criterio. Observamos que para ε=1.0, a menudo A*ε tomaba rutas claramente más largas pero de muy bajo consumo. Siguiendo el ejemplo anterior, con ε=1.0 el algoritmo incluso pudo elegir un camino de 12 pasos evitando totalmente terreno difícil, reduciendo las calorías a quizás 8 (muy por debajo de las 20 iniciales). Sin embargo, esta mayor libertad implica que **se expanden muchos más nodos**: la búsqueda examina numerosas alternativas dentro de ese amplio margen de costo. En nuestras pruebas, el número de nodos explorados con ε=1 llegó a ser más del doble que con ε=0 en algunos mapas, por la cantidad de desvíos evaluados.

En la Figura 3 se muestra la variación típica del **coste** y de las **calorías** de la solución en función de ε (manteniendo el mismo mapa y par de inicio/objetivo para todos los casos):

Como se aprecia, conforme ε aumenta, el coste del camino encontrado (línea azul, en número de pasos) crece gradualmente, mientras que las calorías consumidas (línea roja) tienden a decrecer, alcanzando un punto de saturación donde ya no es posible reducir mucho más el consumo energético sin recorrer distancias excesivas. El usuario o diseñador puede elegir ε según el compromiso deseado entre longitud de la ruta y consumo energético. Por ejemplo, ε=0.2–0.3 podría considerarse un buen balance en ciertos mapas: apenas aumenta un 10–15% la distancia, pero puede ahorrar ~20% de calorías. Valores mayores de ε tienen rendimientos marginales decrecientes en ahorro de calorías, a cambio de rutas mucho más largas.

En cuanto al **rendimiento computacional**, A*ε tiende a explorar más nodos que A* a medida que ε crece, porque mantiene más estados en la lista focal (muchos de los cuales finalmente no serán parte de la solución, pero igualmente se examinan). Esto se observó en las pruebas: para ε=0 (A*), el algoritmo expandió N nodos; con ε=0.5, expandió aproximadamente 1.3*N; y con ε=1.0, llegó a ~2*N (doblando el esfuerzo, coincidiendo con permitir hasta doble de coste). La diferencia no siempre es tan extrema y depende del mapa: si las rutas alternativas tienen costos muy cercanos al óptimo, se explorarán con incluso valores bajos de ε, aumentando nodos explorados. En mapas donde las rutas alternativas son claramente peores, A*ε aún con ε grande no las expandirá hasta agotar otras opciones de menor f, moderando el sobrecoste en cómputo.

Comparación entre A *y A*ε (Coste vs Calorías)

Dado que A *(clásico) se enfoca únicamente en minimizar el costo (distancia) y A*ε introduce una consideración de segundo criterio (calorías), es interesante comparar directamente las soluciones que

produce cada enfoque. A continuación resumimos las diferencias con ayuda de gráficas que confrontan el **coste de la ruta** y las **calorías consumidas**:

En la figura se contrasta, para un mapa de ejemplo, la ruta obtenida por **A** *(ε=0) frente a la ruta de A*ε** con un cierto ε (elegido para mejorar calorías). Podemos observar que:

- La ruta de A *es más corta en distancia (menor número de pasos). Por diseño, A* siempre encuentra o prefiere el camino óptimo en longitud (o costo definido) sin considerar el consumo de terreno. Por ejemplo, en el escenario mostrado la ruta A* tiene longitud 50 (pasos) atravesando una zona pantanosa.

- La ruta de A*ε es algo más larga en número de pasos (por el margen que le da ε), pero a cambio evita zonas de alto costo energético. En el ejemplo, la ruta A*ε rodea parcialmente el pantano aumentando la distancia a 60 pasos, pero logrando reducir las calorías totales de 2000 (ruta A*) a solo 1400.

- En cuanto a **nodos explorados**, A*ε típicamente explorará más (como se discutió), por lo que puede ser menos eficiente computacionalmente. En el ejemplo, A* expandió unos 8000 nodos, mientras A*ε (ε=0.3) expandió ~9500 nodos para encontrar su ruta (un incremento razonable, dado el beneficio obtenido en calorías).

En general, la comparación muestra el **trade-off** claro: **A** *prioriza minimizar la longitud del camino, ignorando el coste del terreno (lo que puede resultar en rutas "baratas" en distancia pero "caras" en energía), mientras que A*ε *permite sacrificar algo de distancia a cambio de mejorar significativamente el coste energético. La decisión de usar uno u otro dependerá de los objetivos: si buscamos la ruta más corta posible usaremos A, pero si nos interesa optimizar otro factor (p.ej. consumo de batería en un robot), A*ε proporciona un mecanismo flexible para equilibrar ambos objetivos.

Pruebas con Weighted A *(A* con Pesos)

Otra variante implementada fue el **Weighted A** *(A* ponderado), donde se introduce un peso *w* que ajusta la influencia de la heurística en la función de evaluación: en lugar de f = g + h, se utiliza **f = (1–w)*g* + *w*h** (en nuestro caso definimos $0 \leq w \leq 1$). Esta fórmula permite sesgar la búsqueda:

- Con *w* cercano a 1, el algoritmo se aproxima a una búsqueda greedy (prioriza fuertemente la heurística, *g* casi no influye).
- Con *w* = 0, se obtiene g + 0*h = g, es decir, una búsqueda de costo uniforme (Dijkstra) sin guía heurística.
- Con valores intermedios (e.g. 0.5), se pondera equitativamente costo recorrido y estimado.

Cabe señalar que nuestra definición es un poco inversa a la habitual (donde *w*>1 multiplica a h); aquí acotamos *w* en [0,1] para interpolar entre búsqueda voraz y uniforme. Para *w*=1 se usa solo h (algoritmo más rápido pero no asegura óptimo), y para *w*=0 se ignora h (búsqueda exhaustiva pero óptima).

Se realizaron pruebas variando *w* para observar su efecto en la eficiencia y en la calidad de la ruta. En las experimentaciones, medimos principalmente el número de **nodos expandidos** y el **coste de la ruta** resultante, comparando con los extremos (A* y Dijkstra). Los resultados se resumen en la Figura 4:

Algunas conclusiones de estas pruebas:

- Un peso **moderado (w ≈ 0.5–0.8)** suele brindar una aceleración notable en la búsqueda sin comprometer la optimalidad en nuestros escenarios. Por ejemplo, con *w*=0.8 en un mapa de tamaño mediano, la cantidad de nodos explorados se redujo alrededor de un 50% en comparación con *w*=0 (búsqueda no informada), mientras que el camino hallado mantuvo el mismo costo óptimo que A *(cuando la heurística es admisible y consistente, valores* w < *1 garantizan optimalidad; en nuestras pruebas* w=*0.8 seguía encontrando la ruta óptima en distancia). Vimos que* w=*0.8 exploró ~11 nodos vs ~16 nodos de A* en un test simple, con igual longitud de solución.

- Con *w* acercándose a **1 (muy heurístico)**, el algoritmo expande aún menos nodos, pero aquí sí observamos potencial de perder la optimalidad. *w*=1.0 equivale a una búsqueda greedy pura (basada solo en h). En un mapa complejo con obstáculos, *w*=1 a veces encontró rutas subóptimas. En un caso de prueba, *w*=1 exploró ~10 nodos (frente a 26 nodos de A*), pero pasó por alto un camino más corto al guiarse únicamente por la heurística (aunque en otro caso afortunado,* w=*1 halló la misma ruta óptima pero con menos costo energético, debido a la heurística llevándolo por terreno ligeramente diferente). En general,* w=*1 debe usarse con precaución ya que no garantiza la solución óptima**.

- Por otro lado, *w* muy bajo (cercano a 0) hace que A *se comporte casi como Dijkstra: asegura el óptimo, pero expande muchos más nodos. En nuestras mediciones,* w=*0 (búsqueda no informada) llegó a expandir hasta ~60% más nodos que A* estándar en mapas con grandes áreas abiertas, lo cual demuestra la importancia de la heurística para enfocar la búsqueda.

En definitiva, Weighted A *ofrece una familia de algoritmos entre los cuales A* clásico es *w*= valor intermedio (equivalente en nuestra fórmula a usar *w* tal que f = g+h; en nuestra implementación exacta, *w*=0.5 produce f = 0.5g+0.5h, que proporcionalmente ordena igual que g+h). Ajustar *w* nos permite **afinar el rendimiento**: valores altos hacen la búsqueda más **rápida** pero potencialmente **subóptima**, valores bajos garantizan **óptimo** pero consumen más tiempo. Una posible estrategia es comenzar con un *w* algo alto para obtener pronto una solución aproximada y luego refinar (aunque ese refinamiento no se implementó en esta práctica, es la idea detrás de algoritmos como A* dinámicamente ponderado).

Para esta práctica, finalmente se dejó *w*=0.8 como valor predeterminado ajustable mediante un slider en la interfaz gráfica, para que el usuario pueda experimentar la diferencia: moviendo el slider hacia 1 la búsqueda se agiliza pero podría saltarse la ruta ideal, hacia 0 se vuelve más completa pero lenta.

Mapas de Prueba y Casos Considerados

Se prepararon y utilizaron varios **mapas de prueba** para validar el comportamiento de los algoritmos en distintas situaciones típicas:

- **Mapa 1: Camino largo con obstáculos dispersos.** Un mapa de tamaño grande (por ejemplo 15x15) con el inicio y el objetivo en esquinas opuestas, y numerosos obstáculos (#) distribuidos. Este caso fuerza al algoritmo a explorar un camino muy **largo** alrededor de los obstáculos. La intención es probar la eficiencia de A *en distancias grandes. Con este mapa comprobamos que A* encuentra correctamente el camino más corto entre extremos (de unas ~25 celdas) incluso cuando hay que bordear muchos obstáculos. A*ε en este entorno no ofrecía mucha diferencia, dado que no había rutas alternativas significativamente diferentes en calorías (los obstáculos obligan casi*

un único camino posible). El énfasis fue en rendimiento: A expandió del orden de cientos de nodos para hallar la ruta, mientras que algoritmos no informados hubieran necesitado miles.

- **Mapa 2: Escenario sin solución.** Se diseñó un mapa donde el objetivo está completamente **rodeado por obstáculos** (un recinto cerrado sin puerta) para verificar que el algoritmo detecta correctamente la ausencia de camino. En este mapa, tras explorar todos los estados alcanzables, la frontera queda vacía y A *termina informando que* no existe un camino válido entre origen y destino*. Efectivamente, la implementación manejó este caso devolviendo* `(None, None)` *como resultado, y la interfaz muestra un mensaje de error y no traza ruta. Esto valida el criterio de parada por fallo. (Un ejemplo: un mapa con un "anillo" de muros # alrededor del objetivo* G*; el algoritmo expandió todos los estados accesibles fuera del anillo –marcados en amarillo como explorados– y no pudo llegar a* G*, confirmando la situación sin solución).

- **Mapa 3: Rutas múltiples (terrenos variables).** En este mapa colocamos tanto terrenos de **agua (~)** como de **roca (*)** creando al menos dos rutas alternativas hacia la meta: una más corta atravesando rocas, y otra un poco más larga pasando por agua o terreno normal. El objetivo es que A *(distancia) tome siempre la ruta más corta, mientras que A*ε pueda preferir la otra si ε lo permite. En concreto, en un mapa de 10x10 situamos el origen y meta de forma que había un camino directo de ~15 pasos cruzando una zona de rocas (con alto coste calórico), versus un camino de ~18 pasos alrededor principalmente por tierra firme. Como se esperaba, A *eligió el camino de 15 pasos por las rocas, consumiendo unas ~90 calorías, mientras que A*ε con ε=0.4 optó por el camino de 18 pasos evitando las rocas y gastando ~60 calorías. Así pudimos visualizar claramente la diferencia de criterios. Este mapa también permitió probar la robustez de la implementación de la lista focal y la correcta contabilización de calorías en distintos terrenos.

- **Mapa 4: Obstáculos formando laberinto.** Adicionalmente, probamos un mapa estilo laberinto angosto donde el algoritmo debe explorar caminos que se bifurcan y algunos son callejones sin salida. Esto puso a prueba la gestión de la lista cerrada (para no re-explorar estados) y la corrección en la reconstrucción del camino. El A* encontró la salida correcta del laberinto, y al visualizar la "matriz de estados explorados" en la interfaz se pudo ver cómo había numerado muchas celdas muertas-end antes de encontrar el camino, lo cual concordó con la lógica esperada.

En conjunto, estos escenarios cubren casos de prueba importantes y nos dieron confianza de que la implementación funciona correctamente bajo distintas condiciones (rutas largas, no solución, múltiples rutas, laberintos).

Instrucciones de Ejecución del Programa

Se ha desarrollado una interfaz gráfica en Python (usando la librería **Pygame**) para poder ejecutar y visualizar cómodamente el funcionamiento de los algoritmos. A continuación se detallan los pasos para ejecutar el programa y utilizar sus funciones, dirigidos a un usuario general sin conocimientos específicos de programación:

1. **Requisitos previos:** Asegúrese de tener instalado Python 3 en su sistema. Además, es necesario instalar la biblioteca *Pygame* para la interfaz (se puede hacer fácilmente ejecutando `pip install pygame` en la terminal).

2. **Lanzamiento de la aplicación:** Ubíquese en el directorio `Fuente` del proyecto (donde reside el código). Ejecute el programa principal con el comando:

```
python main.py
```

Esto cargará por defecto el mapa `mapa.txt`. Alternativamente, puede especificar un mapa distinto como parámetro, por ejemplo: `python main.py mapa2.txt` para cargar el mapa2. También existen parámetros opcionales `--w` (peso de heurística) y `--eps` (epsilon de A*ε) si se desea cambiar sus valores por defecto (w=0.8, ε=0.1).

3. **Ventana de juego:** Al ejecutarse, aparecerá una ventana gráfica titulada "Práctica SI - A". *En ella verá el mapa cargado representado en una cuadrícula. Cada tipo de terreno se muestra con un color distinto: por ejemplo, muros azules (#) como obstáculos, áreas de hierba en beige (.), agua en celeste (~), roca en marrón (*)... En la parte inferior de la ventana se encuentran dos botones ("A" *y "A*ε"), un deslizador de peso *w*, y un texto que mostrará el **Coste** y **Calorías** del camino encontrado.

4. **Seleccionar origen y destino:** Para indicar el punto de partida y la meta, use el **mouse** sobre el mapa:

5. Haga *clic izquierdo* en una celda para colocar el **origen** (aparecerá un icono de conejo). Puede cambiarlo haciendo clic izquierdo en otra celda posteriormente.

6. Haga *clic derecho* en una celda para colocar el **destino** (aparecerá un icono de zanahoria). De igual forma, se puede reubicar con sucesivos clics derechos.
   Note: No se permite colocar origen o meta sobre muros; si lo intenta, el programa avisará por consola "Esa casilla no es válida". Coloque ambos en celdas transitables antes de continuar.

7. **Ajustar peso heurístico (opcional):** El deslizador horizontal rotulado con `w=` en la parte inferior permite ajustar el valor de *w* para Weighted A*. Por defecto inicia en 0.80. Si desea realizar una búsqueda A** clásica, puede colocar *w=0.5* (que equivale en nuestra fórmula a f = 0.5g+0.5h ≈ g+h proporcional). Para enfatizar más la heurística (búsqueda más golosa) acerque el slider hacia 1.00; para hacerla más uniforme (tipo Dijkstra) muévalo hacia 0.00. El valor numérico exacto de w se muestra a la izquierda del deslizador en tiempo real. (Este parámetro afectará solo al algoritmo A* *del botón izquierdo, no al A*ε).

8. **Ejecutar algoritmo A* *o A*ε:** Una vez colocados origen y destino, pulse el botón deseado:

9. Botón **A*** *(izquierdo): ejecuta la búsqueda A** (ponderada según el *w* seleccionado). El programa calculará la ruta más corta y la resaltará.

10. Botón **A*ε** (derecho): ejecuta la búsqueda epsilon. Usará el valor ε indicado en los parámetros (por defecto 0.1 si no se cambió via línea de comandos). Actualmente no hay un control deslizante para ε en la interfaz, por lo que si quiere probar distintos ε debe reiniciar el programa especificando `--eps` distinto. Con el valor dado, el algoritmo calculará la ruta equilibrada en coste-calorías y la mostrará.

11. **Visualización de resultados:** Cuando el algoritmo termine, verá el **camino encontrado dibujado en color amarillo** en el mapa. Además:

12. El origen (conejo) y destino (zanahoria) se mantienen visibles sobre sus celdas.
13. En el texto inferior verá **Coste=X** (número de pasos del camino) y **Calorías=Y** (suma de calorías de todas las celdas recorridas). Estos valores le permiten cuantificar la ruta. Por ejemplo, un coste=10 indica que el camino tiene 10 movimientos; calorías=30 podría indicar que atravesó

principalmente terreno normal (2 por paso, total ~20) con quizás algún paso por agua (4) para sumar 30, etc.

14. Si no se encontró ruta (por ejemplo, si objetivo inaccesible), el coste mostrado será -1 y se imprimirá en la consola un mensaje de "No existe un camino válido". No se coloreará ninguna ruta en ese caso.

15. **Iterar nuevas búsquedas:** Puede mover el origen y destino y volver a ejecutar los algoritmos cuantas veces quiera (incluso alternar A *y A*ε para comparar resultados). Simplemente tras mover conejo o zanahoria, pulse de nuevo el botón deseado. El nuevo resultado sobrescribirá el anterior. Si quiere limpiar la ruta amarilla anterior sin realizar nueva búsqueda, puede hacer clic en cualquier celda (esto resetea la selección de algoritmo actual).

16. **Cerrar la aplicación:** Para salir, haga clic en la **X de la ventana** o presione la tecla **Esc** si la ventana tiene foco. Esto finalizará el programa.

Siguiendo estos pasos, un usuario puede experimentar con diferentes mapas y configuraciones. Por ejemplo, se sugiere probar ajustar *w* al mínimo y al máximo para ver cómo cambia la ruta (y el tiempo de cómputo), o lanzar A*ε con un ε grande para observar una ruta alternativa. La interfaz gráfica facilita visualizar estos cambios de manera intuitiva.

(Nota: En caso de querer añadir nuevos mapas de prueba, simplemente cree un archivo de texto con formato similar a los provistos – usando `# . ~ *` *– y ejecútelo pasándolo como argumento a main.py. El programa leerá automáticamente la matriz de ese fichero.)*

Bibliografía

- **Stuart Russell, Peter Norvig.** *Inteligencia Artificial: Un Enfoque Moderno*. Pearson Educación, 3ª ed., 2010. (Capítulos sobre búsqueda heurística informada, algoritmo A* y variantes)
- **Nils J. Nilsson.** *Principles of Artificial Intelligence*. Tioga Publishing, 1980. (Sección sobre propiedades de óptimo del algoritmo A*)
- **Hart, P.E.; Nilsson, N.J.; Raphael, B.** *"A Formal Basis for the Heuristic Determination of Minimum Cost Paths"*. IEEE Transactions on Systems Science and Cybernetics, 4(2):100-107, 1968. (Artículo clásico que introduce el algoritmo A*)
- **Pohl, I.** *"Heuristic Search Viewed as Path Finding in a Graph"*. Artificial Intelligence, 1(3):193-204, 1970. (Describe variantes como A* con subóptimo controlado)
- **Temario de la asignatura Sistemas Inteligentes (UA)** – Apuntes y transparencias del curso 2024/25, especialmente Tema 3: *"Búsqueda Heurística (A*, algoritmos subóptimos y heurísticas)"* (Autores: Departamento de Ciencia de la Computación e Inteligencia Artificial, UA). [Referencias internas del curso]
