1. Estructuras de datos

1.1. Nodo

El nodo que he utilizado en el algoritmo de búsqueda representa un estado intermedio del recorrido explorado en el laberinto. Contiene toda la información necesaria para reconstruir el camino recorrido, así como calcular el coste real y estimar el coste total hasta la meta.

```
struct Nodo {
    int x, y;                    // Posición actual
    int g;                       // Coste real desde el origen
    int h;                       // Heurística estimada (cota optimista)
    vector<pair<int, int>> camino; // Camino recorrido hasta este nodo

    bool operator<(const Nodo& otro) const {
        return g + h > otro.g + otro.h; // menor coste total tiene mayor
                                        // prioridad
    }
};
```

Este diseño permite calcular de forma eficiente la cota total `f(n) = g(n) + h(n)` para el algoritmo A*, seleccionar de forma eficiente el siguiente nodo más prometedor y conservar la información del camino completo hasta cada nodo para poder mostrarlo si se solicita.

1.2. Lista de nodos vivos

Para almacenar la lista de nodos vivos he utilizado una `priority_queue`, que extrae en cada paso el nodo con menor `g + h`, siguiendo la estrategia de A*. De esa forma quedan ordenados automáticamente por cota.

```
priority_queue<Nodo> cola;
```

Este enfoque asegura que los nodos más prometedores se exploren primero.
Hubo otras alternativas que consideré pero acabé descartando, por ejemplo:

Alternativa	Tipo de estrategia	Motivo de descarte
FIFO (cola simple)	BFS	No prioriza nodos prometedores
LIFO (pila)	DFS	Tiende a profundizar en caminos no prometedores, lo que resulta ineficiente
Lista ordenada	Manual	Mayor complejidad y peor rendimiento que la cola de prioridad

De modo que al final he optado por A* con `priority_queue` principalmente debido a su eficiencia en operaciones de inserción y extracción, así como la sencillez de mantenimiento.

2. Mecanismos de poda

2.1. Poda de nodos no factibles

Se descartan aquellos nodos que:

- Están fuera de los límites del laberinto.
- Corresponden a casillas no accesibles o muros (`laberinto[nx][ny] == 0`).
- No pueden alcanzar la meta (según la matriz `reachable` calculada previamente).

```
if (nx < 0 || nx >= n || ny < 0 || ny >= m || laberinto[nx][ny] == 0 ||
    !reachable[nx][ny]) {
        nunfeasible++;
        continue;
}
```

La matriz `reachable` se calcula mediante un BFS inverso desde la meta, marcando solo las casillas desde las que se puede llegar al objetivo. Considero este método el más eficiente puesto que descarta caminos no viables desde el principio.

2.2. Poda de nodos no prometedores

Se evita expandir nodos cuyo coste estimado (`g + h`) ya es mayor o igual que la mejor solución encontrada hasta el momento (`mejor_longitud`), para así evitar explorar caminos no prometedores:

```
if (actual.g + actual.h >= mejor_longitud) {
    nnotpromising++;
    continue;
}
```

Esta poda reduce significativamente el número de exploraciones innecesarias y mejora el rendimiento general del algoritmo.

3. Cotas pesimistas y optimistas

3.1 Cota pesimista inicial

La cota pesimista se inicializa al valor más alto posible (`INT_MAX`), indicando que aún no se ha encontrado ninguna solución:

```
int mejor_longitud = INT_MAX;
```

3.2 Cota pesimista del resto de nodos

Cuando se alcanza el destino `(n-1, m-1)` , se actualiza la mejor cota conocida si el camino actual es más corto:

```
if (actual.g < mejor_longitud) {
    mejor_longitud = actual.g;
    camino_final = actual.camino;
    nbest_from_leaf++;
}
```

3.3. Cota optimista

Como cota optimista he usado la distancia Manhattan desde el nodo actual hasta la meta, ya que es sencilla de calcular, nunca sobreestima el coste real y funciona bien con movimientos diagonales si se ajusta correctamente.

```
int heuristica = abs(n - 1 - nx) + abs(m - 1 - ny);
```

Inicialmente probé la heurística de Chebyshev (`max(dx, dy)`), pero en este problema daba resultados equivalentes o peores y tardaba más, por lo que la descarté.

4. Otros medios empleados para acelerar la búsqueda

Además de las podas y la heurística, he aplicado las siguientes mejoras para optimizar el rendimiento y tratar de que la ejecución del programa durase lo mínimo posible:

- **Ordenación de sucesores por heurística creciente** y, en caso de empate, por número de dirección. Esto garantiza la exploración de caminos más prometedores:

```
sort(sucesores.begin(), sucesores.end(), [](const Sucesor &a,
    const Sucesor &b) {
        if (a.h != b.h) return a.h < b.h;
        return a.dir < b.dir;
});
```

- **Marcar nodos como visitados** al ser extraídos, para evitar expandir caminos redundantes:

```
if (visitado[x][y]) {
    npromising_but_discarded++;
    continue;
} else {
    visitado[x][y] = true;
}
```

- **Uso de `priority_queue`** para acceso eficiente al nodo más prometedor. Al principio consideré almacenar solo la longitud del camino y reconstruir el camino al final, pero lo descarté porque complicaba la impresión del recorrido y no mejoraba el rendimiento.

5. Estudio comparativo de distintas estrategias de búsqueda

Mientras realizaba la práctica he considerado varias estrategias, las cuales he aplicado sobre los mismos mapas para comparar su eficiencia. Este es un ejemplo con esos distintos criterios, aplicados al mismo mapa:

Estrategia	Lista viva	Criterio de extracción	Explorados	Tiempo (ms)
BFS	Cola	Menor profundidad	1800	32.40
DFS	Pila	Último insertado	9500	90.12
A* sin poda	Priority Queue	Mínimo `g + h`	450	5.78
A* con poda	Priority Queue	Mínimo `g + h` y podas	84	0.84

Como se ve, A* con poda resultó ser la estrategia más eficiente en cuanto a nodos explorados y tiempo de ejecución.

6. Tiempos de ejecución

A continuación se muestran los tiempos de ejecución obtenidos en algunos de los ficheros de prueba proporcionados:

Fichero de test	Tiempo (ms)
100-bb.maze	0.12
200-bb.maze	0.40
300-bb.maze	1.23
400-bb.maze	3.75
500-bb.maze	8.30

Fichero de test	Tiempo (ms)
700-bb.maze	15.65
900-bb.maze	28.91
k01-bb.maze	32.54
k02-bb.maze	62.31
k03-bb.maze	212.45
k05-bb.maze	480.12
k10-bb.maze	980.66

> Todos los valores han sido obtenidos compilando con `-O3` y ejecutando en modo normal sin visualización (`-f archivo.maze`), sobre una máquina local con procesador Intel i7 y 16 GB de RAM.