PONTIFICIA UNIVERSIDAD JAVERIANA
INTRODUCCIÓN A LA INTELIGENCIA ARTIFICIAL
PROYECTO 1

Pontificia Universidad
JAVERIANA
Colombia

Integrantes:

Juliana Sofía Novoa Solano
Miguel Francisco Vargas Contreras

1. Resumen:

Este proyecto se basa en encontrar la solución de un laberinto representado como una matriz NxN encontrando la ruta más óptima de este. Para la solución, se implementó un programa en C++ que transforma el laberinto en un grafo para, posteriormente, aplicar sobre el algoritmos de búsqueda no informada (BFS, DFS y la función de cálculo heucarística) y un algoritmo informado (A*). El documento detalla el proceso de transformación, la implementación de cada algoritmo y el análisis comparativo de los resultados obtenidos.

1.1. Compilación y ejecución

El código solo ha sido probado y compilado en windows 11, se incluye en los archivos del proyecto un script .bat que compila y ejecuta el proyecto.

2. Estructura lectura laberinto:

2.1 Lectura del laberinto

- **Menu**

Primero para efectos de la obtención de datos, se creó un menú para el usuario, véase la imagen 1.

```
int menu( ) {
  int opt = 0;

  do {
    cout << "1. Cargar laberinto" << endl;
    cout << "2. Función en profundidad" << endl;
    cout << "3. Función en anchura" << endl;
    cout << "4. Función heurística" << endl;
    cout << "5. Funcion A*" << endl;
    cout << "6. Salir" << endl;

    cout << "> ";
    cin >> opt;
  } while (opt < 1 || opt > 7);

  return opt;
}
```

Imagen 1. Menu.

- **Carga del laberinto**

Durante el proceso de carga del laberinto surgió la duda entre ingresar los datos manualmente mediante cin o, como se decidió finalmente, utilizar archivos de texto que tuvieran los datos del laberinto previamente escritos. En esta implementación, el único dato solicitado al usuario es el nombre del archivo .txt que contiene el laberinto.

Durante el desarrollo se presentaron algunos inconvenientes relacionados con las validaciones del tamaño, la entrada y la salida del laberinto. Finalmente, el programa lee los valores n y m que definen el tamaño del laberinto y, posteriormente, los datos en forma de matriz, donde:

- El valor 2 representa la entrada del laberinto (único).
- El valor 3 representa la meta o salida(único).
- Los valores 0 corresponden a espacios libres,
- Y los valores 1 indican paredes.

Si la matriz no es demasiado grande y todos los datos son válidos, una vez cargada correctamente, se imprime en pantalla para verificar su contenido.

```
mazeFile >> maze->n;
mazeFile >> maze->m;

if (maze->n < 0 || maze->m < 0) {
  cout << "Matriz con tamaño invalido" << endl;
  return;
}

maze->matrix = new char*[maze->n];  // Filas

for (int i = 0; i < maze->n; i++) {
  // Crear la fila con maze->m elementos
  // *(maze->matrix + i) = new char[maze->m];
  maze->matrix[i] = new char[maze->m];

  for (int j = 0; j < maze->m; j++) {
    char c;
    mazeFile >> c;

    // Validaciones
    if (c != '0' && c != '1' && c != '2' && c != '3') {
      cout << "Ingresaste un valor inválido, intenta de nuevo." << endl;

      delete[] maze->matrix;
      maze->matrix = nullptr;

      return;
    }

    maze->matrix[i][j] = c;
  }
}
```

Imagen 2. Carga de los datos del laberinto.

2.2 Transformación a grafo

Para que el proceso de los algoritmos fueran más eficientes se transformó la matriz cargada a un grafo utilizando lista de adyacencia.

1. **Construcción de aristas:** Se construye una lista de adyacencia que, para cada nodo del grafo, evalúa los posibles movimientos y almacena las posiciones vecinas que sean transitables.

```
void tryAddAdj(Maze* maze, Graph* graph, int i, int j, int di, int dj) {
  if (validPos(maze, i + di, j + dj)) {
    if (maze->matrix[i + di][j + dj] == '1') return;

    graph->adj[j + (i * maze->m)].push_back((j + dj) + ((i + di) * maze->m));
  }
}
```

Imagen 3. Función construcción de aristas.

2. **Construcción del grafo:**

- **Inicialización:** Se inicializa el grafo creando una instancia de la estructura de este, inicialización de los punteros de inicio y salida y creación de la lista de adyacencia de vectores para guardar los IDs de los vecinos conectados.

```
Graph* graph = new Graph;

  graph->start = nullptr;
  graph->end = nullptr;

  long node_count = maze->m * maze->n;

  graph->maze = maze;
  graph->nodes = vector<Node*>(node_count);
  graph->adj = vector<vector<int>>(node_count);
```

Imagen 4. Inicialización Grafo.

- **Recorrido de la matriz:** Utilizado para mapear cada celda de la matriz a un nodo del grafo.

```
for (int i = 0; i < maze->n; i++) {
    for (int j = 0; j < maze->m; j++) {
```

Imagen 5. Bucle recorrido matriz.

- **Creación del nodo:** Por cada celda se crea un objeto *Node* para representar una celda en el laberinto, primero a cada nodo se le asigna un ID único, se guardan las coordenadas originales de este para saber su posición dentro del laberinto y reconstruir su camino después. Se guarda el tipo de celda según su valor (salida, meta, pared o espacio). Finalmente, con el ID del nodo, se guarda en el grafo.

```
Node* node = new Node;

      node->id = j + (i * maze->m);
      node->y = i;
      node->x = j;
      node->type = maze->matrix[i][j];

      graph->nodes[node->id] = node;
```

Imagen 6. Creación de nodos.

- **Identificación de salida y meta:** Usando condicionales se valida la salida y meta del laberinto (2 para la salida) y (3 para la meta). Manejando que no hallan más de una salida o meta para el laberinto.

```
if (node->type == '2') {
        if (graph->start != nullptr) {
          cout << "Grafo invalido, multiples nodos de inicio" << endl;

          delete graph;
          return nullptr;
        }
        graph->start = node;
      }
      if (node->type == '3') {
        if (graph->end != nullptr) {
          cout << "Grafo invalido, multiples nodos de salida" << endl;

          delete graph;
          return nullptr;
        }
        graph->end = node;
      }
```

Imagen 7. Identificación salida y meta para el grafo.

- **Generación de aristas:** En esta función del programa hubieron algunos desafíos, ya que no se estaban teniendo en cuenta los nodos de salida y entrada como casillas válidas. Haciendo ya la implementación correcta, en el bucle for, utilizando la función "*tryAddAdj*" vease la Imagen 3. Si la

casilla es válida, (tiene valores 0, 2 o 3). Se revisan sus vecinos y se guardan en el grafo los vecinos cuyos valores también sean válidos, transitables diferentes de 1.

```
if (node->type == '0' || node->type == '2' || node->type == '3') {
        // Pos Arriba
        tryAddAdj(maze, graph, i, j, -1, 0);
        // Pos Derecha
        tryAddAdj(maze, graph, i, j, 0, 1);
        // Pos Abajo
        tryAddAdj(maze, graph, i, j, 1, 0);
        // Pos Izquierda
        tryAddAdj(maze, graph, i, j, 0, -1);
      }
```

Imagen 8. Generación de aristas.

- Finalmente se imprime mensaje de confirmación o error al usuario.

3. Algoritmos

3.1 Profundidad

Para este algoritmo se utilizó backtracking en una función recursiva, inicialmente se marca el nodo que se está revisando cómo visitado para evitar ciclos infinitos, por cada nodo se revisan a todos los nodos conectados a él, se revisa si estan visitados y si no significa que pueden ser explorados, para explorarlo se guarda al nodo “padre” para luego poder recrear el camino que se recorrió hasta la meta.

La función se llama a sí misma de manera recursiva para seguir explorando desde el vecino que se está revisando, esto iterativamente hasta encontrar la meta del laberinto.

```
bool dfsHelper(int u, int target, Graph* graph, vector<bool>&
visited,
               map<int, int>& parent) {
  visited[u] = true;

  if (u == target) return true;

  for (int v : graph->adj[u]) {
    if (!visited[v]) {
      parent[v] = u;
      if (dfsHelper(v, target, graph, visited, parent)) {
        return true;
      }
    }
  }
  return false;
}
```

Imagen 9. Recorrido algoritmo en profundidad.

Se definió a la función "*ExecuteDfs*" la cual es la que devuelve un puntero al vector de posiciones, que será el camino encontrado hasta la meta. Para esto primero verifica que exista el grafo, luego guarda los valores de la salida y meta.

```
vector<Position>* ExecuteDFS(Execution* execution) {
  if (!execution->graph) {
    return nullptr;
  }

  Graph* g = execution->graph;
  int startId = g->start->id;
  int endId = g->end->id;
```

Imagen 10. Función ExecuteDFS.

Inicialización de estructuras necesarias para el DFS:

1. Vector para marcar los nodos ya visitados (*visited*).
2. Mapa para reconstruir el camino (*parent*) guarda el nodo de partida.
3. Vector de posiciones con la ruta encontrada (*path*).

```
vector<bool> visited(g->nodes.size(), false);
  map<int, int> parent;
  vector<Position>* path = new vector<Position>();
```

Imagen 11. Estructuras DFS.

Para la reconstrucción del camino se utiliza la función "*dfsHelper*", véase la imagen 9, la cual hace la búsqueda en profundidad, esta retorna si encontró o no la meta. Luego se reconstruye el camino desde la meta al inicio y ya que este se hizo de atrás hacía delante, se invierte para que quede correctamente. Finalmente se devuelve al vector con el camino.

```
if (dfsHelper(startId, endId, g, visited, parent)) {
    int curr = endId;
    while (curr != startId) {
      path->push_back({g->nodes[curr]->y, g->nodes[curr]->x});
      curr = parent[curr];
    }
    path->push_back({g->nodes[startId]->y, g->nodes[startId]->x});

    reverse(path->begin(), path->end());

    return path;
  } else {
    return nullptr;
  }
```

Imagen 12. Reconstrucción del camino DFS.

Desafíos: En la construcción del algoritmo se nos dificulto evitar bucles infinitos sobre nodos ya visitados, se soluciono creando una estructura que marcará a los nodos ya visitados. Otro problema que surgió fue en la reconstrucción del camino ya que estaba creándose a la inversa inicialmente, para arreglarlo se invirtió.

3.2 Anchura

Se inicializar las siguientes estructuras de datos necesarias para la construcción del BFS:

1. Vector para marcar a los nodos ya procesados.
2. Vector que guarda el ID del nodo padre de cada nodo visitado.
3. La cola con los ID de los nodos que faltan por explorar.

```
vector<bool> visited(execution->graph->maze->m *
execution->graph->maze->n,
                     false);
  vector<long> parent(execution->graph->maze->m *
execution->graph->maze->n,
                    -1);

  queue<long> q;
```

Imagen 13. Inicialización estructuras de datos BFS.

La función principal del algoritmo, se basa en inicialmente sacar el nodo que lleva más tiempo en la cola, se revisa todos los vecinos conectados a ese nodo en la lista de adyacencia. Si el vecino fue visitado se ignora, si no, se marca al nodo como visitado, se marca al nodo padre del nodo actual y se mete al vecino en la cola para explorar sus propios vecinos más tarde.

```
q.push(execution->graph->start->id);
  parent[execution->graph->start->id] = -1;
  visited[execution->graph->start->id] = true;

  while (!q.empty()) {
    long curr = q.front();
    q.pop();

    for (long u : execution->graph->adj[curr]) {
      if (visited[u]) continue;
      visited[u] = true;
      parent[u] = curr;

      q.push(u);
```

Imagen 14. Función principal BFS.

Una vez la cola esté vacía o se encuentra la meta, el algoritmo reconstruye al camino desde el final hacía atrás, siguiendo el rastro de padres hasta llegar al inicio, para retornar el resultado se utilizaron iteradores reversos, *rbegin* y *rend* para darle la vuelta y devolver el camino en el orden correcto.

```
long curr = execution->graph->end->id;
  vector<Position> path;

  while (curr != -1) {
    Position pos;
    pos.y = curr / execution->graph->maze->m;
    pos.x = curr % execution->graph->maze->m;

    path.push_back(pos);
    curr = parent[curr];
  }

  return new vector<Position>(path.rbegin(), path.rend());
```

Imagen 15. Reconstrucción del recorrido BFS.

Desafíos: En la construcción del algoritmo se nos dificulto evitar bucles infinitos sobre nodos ya visitados, se soluciono creando una estructura que marcará a los nodos ya visitados. Otro problema que surgió fue en la reconstrucción del camino ya que estaba creándose a la inversa inicialmente, para arreglarlo se invirtió.

3.3 Función de Cálculo Heurística y A*

Para la implementación de estos algoritmos primero se definió a una función que calcula la distancia Manhattan desde la celda (i, j) hasta la que está marcada como la meta. Esa distancia es la heurística la cual A* usa para estimar cuánto falta para llegar a la meta.

```
long calc_heuristic(Execution* execution, int i, int j) {
  return abs(execution->graph->end->y - i) + abs(execution->graph->end->x - j);
}
```

Imagen 16. Cálculo de la heurística.

Carga de datos del laberinto: Tamaño, inicio y meta.

```
int n = execution->graph->maze->n;
  int m = execution->graph->maze->m;

  long start_id = execution->graph->start->id;
  long end_id = execution->graph->end->id;
```

Imagen 17. Carga de datos heurística y A.*

Para pre-calcular la distancia estimada de cada celda vacía (aquellas marcadas con 0) hasta el final se creó la matriz heurística. Para poder implementar a A* se usó una *priority_queue* con *greater* para que actúe como min-heap: el par con menor first sale primero. Finalmente en *weight* se guarda el coto acumulado (g) para llegar a cada celda. Inicializado con un valor muy alto.

```
vector<vector<int>> heuristic(n, vector<int>(m, INFINITY));

  for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
      if (execution->graph->maze->matrix[i][j] == '0')
        heuristic[i][j] =  // Distancia manhattan
            calc_heuristic(execution, i, j);
    }
  }

  priority_queue<pair<long, long>, vector<pair<long, long>>,
                 greater<pair<long, long>>>
      pq;

  vector<vector<long>> weight(n, vector<long>(m, 1e9));

  vector<long> parent(m * n, -1);
```

Imagen 18. Preparación datos.

Para inicializar al nodo principal se mete al inicio en la cola con prioridad (f = g + h) al inicio g = 0, así solo cuenta la heurística, se marca que no tiene padre y se le establece un costo inicial de 0.

```
pq.push(make_pair(heuristic[start_id / m][start_id % m], start_id));
  parent[start_id] = -1;
  weight[start_id / m][start_id % m] = 0;
```

Imagen 19. Inicialización nodo inicial.

En el algoritmo, en cada iteración se extrae de la cola de prioridad el nodo con el menor valor de f = g + h, que representa el camino más prometedor hasta el momento. A partir de ese nodo, identificado como (i, j), se verifica si corresponde al nodo objetivo, en ese caso, el algoritmo finaliza porque ya se ha alcanzado la meta. Si no, se recorren todos los vecinos conectados a ese nodo. Para cada vecino, se calcula un nuevo costo acumulado, ya que en este laberinto cada movimiento tiene un costo uniforme de

1. Si el nuevo costo no mejora el camino previamente registrado hacia ese vecino, se ignora, de lo contrario, se actualiza el peso con el nuevo valor y se registra al nodo actual como su padre.

Finalmente, el vecino se vuelve a insertar en la cola con su nueva prioridad, que corresponde a la suma del costo recorrido (g) y la heurística (h). De esta forma, el algoritmo A* explora los caminos en orden de prioridad, avanzando siempre hacia las rutas que parecen más cercanas a la meta según la heurística Manhattan, la cual es admisible porque nunca sobreestima la distancia real en un movimiento con cuatro direcciones y costo unitario.

```
while (!pq.empty()) {
    pair<long, long> curr = pq.top();
    pq.pop();
    int i = curr.second / m;
    int j = curr.second % m;

    if (curr.second == end_id) break;

    for (long u : execution->graph->adj[curr.second]) {
      int ui = u / m;
      int uj = u % m;

      long n_weigth = weight[i][j] + 1;

      if (weight[ui][uj] <= n_weigth) continue;

      weight[ui][uj] = n_weigth;
      parent[u] = curr.second;

      pq.push(make_pair(weight[ui][uj] + heuristic[ui][uj], u));
    }
  }
```

Imagen 20. Implementación heurística y A.*

Finalmente para la reconstrucción del camino se utiliza al arreglo *parent* para retroceder desde el final, hasta el inicio y se invierte el vector al final para que el camino vaya de inicio a fin.

```
long curr = execution->graph->end->id;
  vector<Position> path;

  while (curr != -1) {
    Position pos;
    pos.y = curr / m;
    pos.x = curr % m;

    path.push_back(pos);
    curr = parent[curr];
  }

  return new vector<Position>(path.rbegin(), path.rend());
  }
```

Imagen 21. Reconstrucción del camino.

Desafíos: Definir la función Manhattan. Tuvimos el mismo problema en la reconstrucción del camino ya que estaba creándose a la inversa inicialmente, para arreglarlo se invirtió.

4. Segunda entrega: Modificaciones

Se intentó reciclar parte del código inicial que teníamos, pero se tuvo que rehacer totalmente la estructura principal del sistema, cambiando la idea de tener una lista de adyacencia general en el grafo a una lista de adyacencia en cada nodo.

Tuvimos problemas al complicarnos demasiado a la hora de colapsar el grafo (eliminar los nodos de “pasillo”), intentamos realizar el descubrimiento uno a uno de los nodos de decisión, pero al final optamos por marcar inicialmente todos los nodos con un número diferente a 2 de entradas y salidas, y también al nodo inicial y al nodo final.

Con esta lista de nodos, hacemos una exploración tipo dfs hasta encontrar otro nodo, así definimos una adyacencia.

4.1 Conversión del Laberinto a un Grafo optimizado

El archivo MatrixToGraph.cxx contiene las funciones que transforman la matriz del laberinto en un grafo optimizado, donde solo se crean nodos de decisión (inicio, meta, bifurcaciones o callejones sin salida).

Esto permite que los algoritmos de búsqueda trabajen más rápido al ignorar los corredores lineales.

```
bool validPos(Maze* maze, int i, int j) {
  if (i < 0 || i >= maze->n) return false;
  if (j < 0 || j >= maze->m) return false;
  return true;
}
```

Verifica si una posición (i, j) se encuentra dentro de los límites del laberinto.

```
bool walkable(Maze* maze, int i, int j) {
  return validPos(maze, i, j) && maze->matrix[i][j] != '1';
}
```

Determina si una celda es transitable, es decir, si no es una pared ('1').

```
int getNCount(Maze* maze, int i, int j) {
  int count = 0;

  // Arriba
  if (walkable(maze, i - 1, j)) count++;
  // Derecha
  if (walkable(maze, i, j + 1)) count++;
  // Abajo
  if (walkable(maze, i + 1, j)) count++;
  // Izquierda
  if (walkable(maze, i, j - 1)) count++;

  return count;
}
```

Cuenta los vecinos transitables de una celda.
Si el resultado es diferente de 2, la celda se considera un nodo de decisión.

```
void getAdj(Maze* maze, vector<vector<Node*>>& nodeMatrix, Node* curr,
            vector<vector<bool>>& visited, int i, int j, int w) {
  if (visited[i][j]) return;
  visited[i][j] = true;

  if (!(i == curr->y && j == curr->x)) {
    if (nodeMatrix[i][j] != nullptr) {
      curr->adj.push_back({w, nodeMatrix[i][j]});
      return;
    }
  }

  if (walkable(maze, i - 1, j))
    getAdj(maze, nodeMatrix, curr, visited, i - 1, j, w + 1);
  if (walkable(maze, i, j + 1))
    getAdj(maze, nodeMatrix, curr, visited, i, j + 1, w + 1);
  if (walkable(maze, i + 1, j))
    getAdj(maze, nodeMatrix, curr, visited, i + 1, j, w + 1);
  if (walkable(maze, i, j - 1))
    getAdj(maze, nodeMatrix, curr, visited, i, j - 1, w + 1);
}
```

Explora el laberinto desde un nodo hasta encontrar otro nodo de decisión.
Cada vez que encuentra uno, crea una arista entre ambos con el peso del corredor (w).

Función Principal:

1. Recorre toda la matriz para crear nodos solo en bifurcaciones, inicio y meta.
2. Conecta esos nodos entre sí usando la función recursiva getAdj.
3. Cada arista almacena el peso del corredor (cantidad de pasos entre nodos).
4. Imprime una versión del laberinto donde los nodos de decisión aparecen como #.

```cpp
Graph* MatrixToGraph(Maze* maze) {
  Graph* graph = new Graph();
  graph->maze = maze;

  vector<Node*> nodes;
  vector<vector<Node*>> nodeMatrix(maze->n, vector<Node*>(maze->m, nullptr));

  // Identificación de nodos de decisión
  for (int i = 0; i < maze->n; i++) {
    for (int j = 0; j < maze->m; j++) {
      Node* node = nullptr;
      switch (maze->matrix[i][j]) {
        case '0': {
          int adjCount = getNCount(maze, i, j);
          if (adjCount != 2) node = new Node();
          break;
        }
        case '2': node = new Node(); graph->start = node; break;
        case '3': node = new Node(); graph->end = node; break;
      }

      if (node != nullptr) {
        node->type = maze->matrix[i][j];
        node->x = j;
        node->y = i;
        nodes.push_back(node);
        nodeMatrix[i][j] = node;
      }
    }
  }

  // Conexión entre nodos de decisión
  for (Node* node : nodes) {
    vector<vector<bool>> visited(maze->n, vector<bool>(maze->m, false));
    getAdj(maze, nodeMatrix, node, visited, node->y, node->x, 0);
  }

  cout << "Laberinto válido, grafo generado" << endl;
  return graph;
}
```

4.2 Algoritmos

1. **DFS**

El archivo DFS.cxx implementa el algoritmo de búsqueda en profundidad (Depth-First Search). Este algoritmo explora el laberinto avanzando lo más profundo posible por cada camino antes de retroceder, utilizando recursión para el recorrido.

```
PathNode* dfsHelper(vector<vector<bool>>& visited, PathNode* curr) {
  visited[curr->node->y][curr->node->x] = true;

  if (curr->node->type == '3') return curr;

  for (pair<int, Node*> u : curr->node->adj) {
    if (visited[u.second->y][u.second->x]) continue;

    PathNode* res = dfsHelper(visited, new PathNode(u.second, curr));

    if (res != nullptr) return res;
  }
  return nullptr;
}
```

Esta función realiza la exploración recursiva del grafo.
En cada llamada, marca el nodo actual como visitado y revisa si corresponde a la meta (type == '3').
Si no es la meta, continúa explorando cada nodo adyacente no visitado.

```
vector<Node*>* ExecuteDFS(Execution* execution) {
  if (!execution->graph) {
    return nullptr;
  }

  int n = execution->graph->maze->n;
  int m = execution->graph->maze->m;

  vector<vector<bool>> visited(n, vector<bool>(m, false));

  Graph* g = execution->graph;
  PathNode* start = new PathNode(g->start, nullptr);
  PathNode* end = dfsHelper(visited, start);


  PathNode* curr = end;
  vector<Node*>* path = new vector<Node*>();

  while (curr != nullptr) {
    path->push_back(curr->node);
    curr = curr->parent;
  }

  reverse(path->begin(), path->end());

  return path;
}
```

Esta función inicializa las estructuras necesarias y llama a dfsHelper para realizar la búsqueda en profundidad. Primero se crean las estructuras visited y el nodo inicial PathNode, que apunta al punto de partida del laberinto. Luego, dfsHelper se encarga de recorrer el grafo marcando las celdas exploradas

hasta encontrar la meta, avanzando de forma recursiva. Una vez alcanzada la meta, el recorrido se reconstruye utilizando los punteros parent de cada PathNode, enlazando los nodos visitados desde la meta hasta el inicio. Finalmente, el vector path se invierte para mostrar el recorrido correcto desde el punto de inicio (2) hasta la salida (3).

2. BFS

El archivo BFS.cxx implementa el algoritmo Breadth-First Search (BFS), el cual explora el laberinto por niveles, garantizando encontrar la ruta más corta entre el punto de inicio (2) y la meta (3).

```
vector<Node*>* ExecuteBFS(Execution* execution) {
  if (!execution->graph) {
    return nullptr;
  }

  int n = execution->graph->maze->n;
  int m = execution->graph->maze->m;

  PathNode* start = new PathNode(execution->graph->start, nullptr);
  PathNode* end = nullptr;

  vector<vector<bool>> visited(n, vector<bool>(m, false));

  queue<PathNode*> q;

  q.push(start);
  visited[start->node->y][start->node->x] = true;

  while (!q.empty()) {
    PathNode* curr = q.front();
    q.pop();

    if (curr->node->type == '3') {
      end = curr;
      break;
    }

    for (pair<int, Node*> u : curr->node->adj) {
      if (visited[u.second->y][u.second->x]) continue;
      visited[u.second->y][u.second->x] = true;

      q.push(new PathNode(u.second, curr));
    }
  }turn path;
}
```

La función inicializa la cola queue<PathNode*> y marca el nodo inicial como visitado. Luego, explora los nodos vecinos por niveles, añadiendo cada nuevo nodo alcanzable a la cola hasta encontrar la meta (type == '3').

```
PathNode* curr = end;
  vector<Node*>* path = new vector<Node*>();

  while (curr != nullptr) {
    path->push_back(curr->node);
    curr = curr->parent;
  }

  reverse(path->begin(), path->end());

  return path;
```

Cuando la salida se encuentra, se reconstruye el camino usando los punteros parent de cada nodo para conectar los pasos en orden inverso.
Finalmente, el vector path se invierte para mostrar la ruta correcta desde el inicio hasta la meta.

3. AStar

El archivo AStar.cxx implementa el algoritmo de búsqueda A* (A estrella), una técnica informada que combina la exploración de BFS con una heurística para priorizar los caminos más prometedores. Este algoritmo garantiza encontrar el camino más corto de forma más eficiente que BFS al estimar la distancia restante hacia la meta.

```
PathNode* start = new PathNode(execution->graph->start, nullptr);
  PathNode* end = nullptr;
  vector<vector<int>> heuristic(n, vector<int>(m, INFINITY));
```

Inicialización de estructuras: Se crea un nodo inicial start que apunta al punto de partida del laberinto. Se inicializa la matriz heuristic con valores infinitos. Esta almacenará los valores h(n) para cada celda.

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    if (execution->graph->maze->matrix[i][j] == '0')
      heuristic[i][j] =  // Distancia manhattan
          calc_heuristic(execution, i, j);
  }
}
```

Se recorre la matriz del laberinto para calcular la heurística solo en las celdas transitables ('0')

```
priority_queue<pair<long, PathNode*>, vector<pair<long, PathNode*>>,
               greater<pair<long, PathNode*>>> pq;

vector<vector<long>> weight(n, vector<long>(m, 1e9));
pq.push(make_pair(heuristic[start->node->y][start->node->x], start));
weight[start->node->y][start->node->x] = 0;
```

1. Se utiliza una cola de prioridad (priority_queue) que siempre extrae el nodo con el menor costo total f.

2. weight guarda los costos acumulados (g) desde el inicio hasta cada celda.

3. El valor inicial 1e9 (infinito aproximado) indica que esas celdas aún no han sido exploradas.

4. Se inserta el nodo inicial en la cola con su valor f = g + h.

```
while (!pq.empty()) {
    pair<long, PathNode*> curr = pq.top();
    pq.pop();

    if (curr.second->node->type == '3') {
      end = curr.second;
      break;
    }

    for (pair<int, Node*> u : curr.second->node->adj) {
      long n_weigth =
          weight[curr.second->node->y][curr.second->node->x] + u.first;

      if (weight[u.second->y][u.second->x] <= n_weigth) continue;

      weight[u.second->y][u.second->x] = n_weigth;
      PathNode* nu = new PathNode(u.second, curr.second);

      pq.push(make_pair(weight[u.second->y][u.second->x] +
                            heuristic[u.second->y][u.second->x],
                        nu));
    }
  }
```

1. Extracción del nodo actual:
   Se toma el nodo con el menor valor f de la cola (el más prometedor).

2. Verificación de la meta:
   Si el nodo actual es la salida ('3'), se detiene el algoritmo.

3. Expansión de vecinos:

- Para cada vecino adyacente, se calcula el nuevo costo acumulado n_weight = g(actual) + costo_arista.
- Si el nuevo costo es menor al anterior, se actualiza el valor en weight.
- Se crea un nuevo PathNode con su referencia al nodo padre y se inserta en la cola con el nuevo valor f = g + h.

Este proceso se repite hasta encontrar el camino más eficiente hacia la meta.

4.3 Reconstrucción del Camino

```
static bool walkable(Maze* maze, int y, int x) {
  if (y < 0 || y >= maze->n || x < 0 || x >= maze->m) return false;
  return maze->matrix[y][x] != '1';
}
```

Verifica si una celda (y, x) es válida y transitable.

Devuelve false si está fuera de los límites o si contiene un muro ('1').

Función Principal:

```
vector<pair<int,int>> expandPath(vector<Node*>* macroPath, Execution* execution)
```

Recibe el camino simplificado (macroPath) y reconstruye todas las celdas intermedias del recorrido en el laberinto original.

```
for (size_t i = 0; i + 1 < macroPath->size(); i++) {
    Node* curr = (*macroPath)[i];
    Node* next = (*macroPath)[i + 1];

    fullPath.push_back({curr->y, curr->x});
```

La función recorre cada par consecutivo de nodos de decisión, tratando de reconstruir el camino que los conecta dentro del laberinto.

Cada conexión se procesa de manera independiente usando un BFS local, garantizando que se encuentren los pasos exactos entre ambos puntos.

BFS para conectar dos nodos:

```
queue<pair<int,int>> q;
    map<pair<int,int>, pair<int,int>> parent;
    q.push({curr->y, curr->x});
    parent[{curr->y, curr->x}] = {-1, -1};
```

1. Se usa una cola (queue) para explorar las celdas adyacentes desde el nodo actual.

2. El mapa parent almacena los predecesores de cada celda, lo que permitirá reconstruir el camino más adelante.

3. Se inicia la búsqueda desde la celda actual (curr).

```
while (!q.empty() && !found) {
      auto [y, x] = q.front();
      q.pop();

      // Revisar vecinos
      for (auto [dy, dx] : vector<pair<int,int>>{{-1,0},{1,0},{0,-1},{0,1}}) {
        int ny = y + dy, nx = x + dx;
        if (!walkable(maze, ny, nx)) continue;
        if (parent.count({ny, nx})) continue;
        parent[{ny, nx}] = {y, x};
        if (ny == next->y && nx == next->x) {
          found = true;
          break;
        }
        q.push({ny, nx});
      }
    }
```

1. Extrae la celda actual de la cola.

2. Revisa sus 4 vecinos (arriba, abajo, izquierda, derecha).

3. Si un vecino es transitable y no fue visitado, lo agrega a la cola.

4. Cuando encuentra la celda destino (next), se detiene la búsqueda.

Desafío: El principal desafío fue reconstruir los pasos intermedios sin hacer el algoritmo demasiado pesado. Una opción inicial era guardar todas las posiciones de cada corredor dentro del grafo, pero eso habría consumido mucha memoria y ralentizado el programa. La solución elegida fue usar un BFS local entre cada par de nodos de decisión, calculando los pasos solo cuando se necesita reconstruir la ruta final.

En general no tuvimos problemas en la re implementación que tuvimos que hacer de cada algoritmo más de tener que re implementarlo con la nueva estructura que tenemos de lista de adyacencia dentro del nodo, en el caso de A* solo agregamos la suma del peso del camino.