tarea-04-grafos

Integrantes: Joaquín Mercado y Mathias Castro

Análisis de Complejidad Algorítmica y Espacial para **Astar (A*):**

```
149     void a_star(Graph &graph) {
150         std::unordered_map<Node *, Node *> parent;
151 
152         // g: distancia desde el origen
153         std::unordered_map<Node*, double> g_score;
154 
155         // f: g + heuristica
156         std::unordered_map<Node*, double> f_score;
157 
158         // min-heap de nodos a visitar, ordenados por f_score
159         //(greater se usa para que el menor este arriba)
160         std::priority_queue<Entry, std::vector<Entry>, std::greater<Entry>> open_set;
161 
162         // Set de nodos ya procesados
163         std::unordered_set<Node*> closed_set;
164 
165         auto heuristic = [this](Node* node) -> double {
166             float dx = node->coord.x - dest->coord.x;
167             float dy = node->coord.y - dest->coord.y;
168             return std::sqrt(dx * dx + dy * dy);
169         };
170 
171         for (auto& [id, node] : graph.nodes) {
172             g_score[node] = std::numeric_limits<double>::max();
173             f_score[node] = std::numeric_limits<double>::max();
174         }
175 
176         g_score[src] = 0.0;
177         f_score[src] = heuristic(src);
178         open_set.push({src, f_score[src]});
179         parent[src] = nullptr;
180 
181         int iterations = 0;
182         while (!open_set.empty()) {
183             Entry current_entry = open_set.top();
184             open_set.pop();
185             Node* current = current_entry.node;
186 
187             // si el nodo ya fue procesado, saltarlo (puede haber duplicados en el heap)
188             if (closed_set.find(current) != closed_set.end()) {
189                 continue;
190             }
191 
192             // marcar el nodo como procesado
193             closed_set.insert(current);
194 
195             iterations++;
196             /*
197             if (iterations % 1000 == 0) {
198                 std::cout << "A* iteration " << iterations << ", open set size: " << open_set.size() << std::endl;
199             }
200             */
201 
202             if (current == dest) {
203                 std::cout << "A* llego a su destino despues de " << iterations << " iteraciones" << std::endl;
204                 break;
205             }
206 
207             for (Edge* edge : current->edges) {
208                 Node* neighbor = nullptr;
```

Para realizar el análisis algorítmico, este se va a realizar por partes (el render no es parte del análisis, ni el dibujo).

De las líneas de código 150 hasta 170, su complejidad algorítmica es O(1), siendo la creación de variables y el cálculo de la heurística.

Pasando a las líneas de código 170 a 175, se inicializan las tablas hash g_score (costo real) y f_score (costo acumulado), siendo su costo algorítmico O(V) y coste espacial de O(V) (g_score, f_score y parent guardan un valor por nodo).

```
    for (auto& [id, node] : graph.nodes) {
        g_score[node] = std::numeric_limits<double>::max();
        f_score[node] = std::numeric_limits<double>::max();
    }
```

Luego se pasa al while. Cabe mencionar que para la inicialización de un set begin() es O(1), el erase(iterator) es O(log N), erase(value) es O(log N) (se realiza búsqueda y borrado) y el insert es O(log N). N es el tamaño del open set, donde **su peor caso va a ser O(V)**.

Sobre el bucle principal (while loop), este procesa nodos hasta que open_set este vacío, en el peor de los casos, el número de iteraciones es el número de nodos, es decir O(V).

En cada iteración: se extrae el nodo con el menor valor de f del conjunto open_set, para ello se utiliza un priority queue (min heap) que tiene una complejidad de extracción de O(log V).

```
181         int iterations = 0;
182         while (!open_set.empty()) {
183             Entry current_entry = open_set.top();
184             open_set.pop();
185             Node* current = current_entry.node;
186
```

Luego se procesan los vecinos del nodo actual, la complejidad de este depende del grado del nodo, O(d), sea *d* el grado del nodo.

```
for (Edge* edge : current->edges) {
    Node* neighbor = nullptr;
    if (edge->src == current) {
        neighbor = edge->dest;
    } else if (!edge->one_way && edge->dest == current) {
        neighbor = edge->src;
    }
```

Si un vecino se actualiza, este se inserta en open_set, con complejidad O(log V):

```
if (tentative_g_score < g_score[neighbor]) {
    parent[neighbor] = current;
    g_score[neighbor] = tentative_g_score;
    f_score[neighbor] = g_score[neighbor] + heuristic(neighbor);

    open_set.push({neighbor, f_score[neighbor]});

    visited_edges.push_back(sfLine(
        current->coord,
        neighbor->coord,
        sf::Color(100, 255, 100, 100),
        1.0f
    ));

    render();
}
```

Calculando el costo total por iteración:

1. Se extraen del nodo de open_set: O(log V)
2. Se iteran sobre sus vecinos con grado d: O(d)
3. Inserción en open_set: O(log V)

El número total de iteraciones es O(V) en el peor caso; sin embargo, cada iteración procesa O(d) aristas, por lo que **el costo total del algoritmo es O(E log V).**

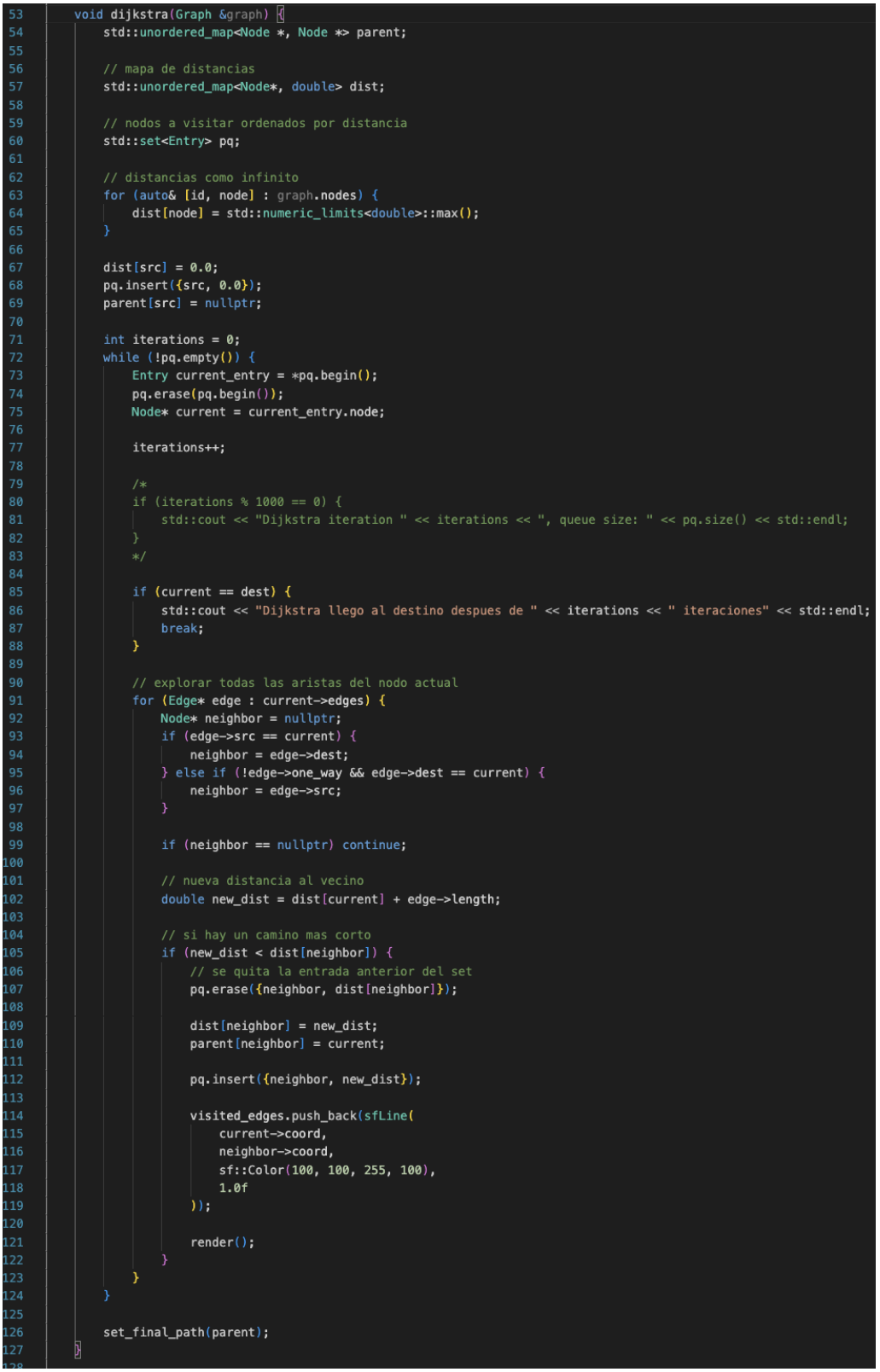
Nota: Si el grafo es no denso, la complejidad de A* se ve afectada. Esto es porque el número de aristas E es mucho menor que el número máximo posible de aristas V^2. En este caso, la complejidad algorítmica se aproxima más a O(V logV), ya que E es mucho menor que V^2 y está más cerca de O(V) en grafos dispersos.

Sobre la complejidad espacial:

1. g_score, f_score y parent son tablas hash que almacenan un valor para cada nodo, este requiere O(V) espacio en memoria.
2. open_set es un min heap que almacena nodos con sus valores f; en el peor caso, el conjunto tiene todos los nodos, por lo que es O(V) espacio en memoria.
3. closed_set es un conjunto que almacena los nodos procesados. En el peor caso, puede contener a todos los nodos, por lo que es O(V)

Complejidad espacial: O(V)

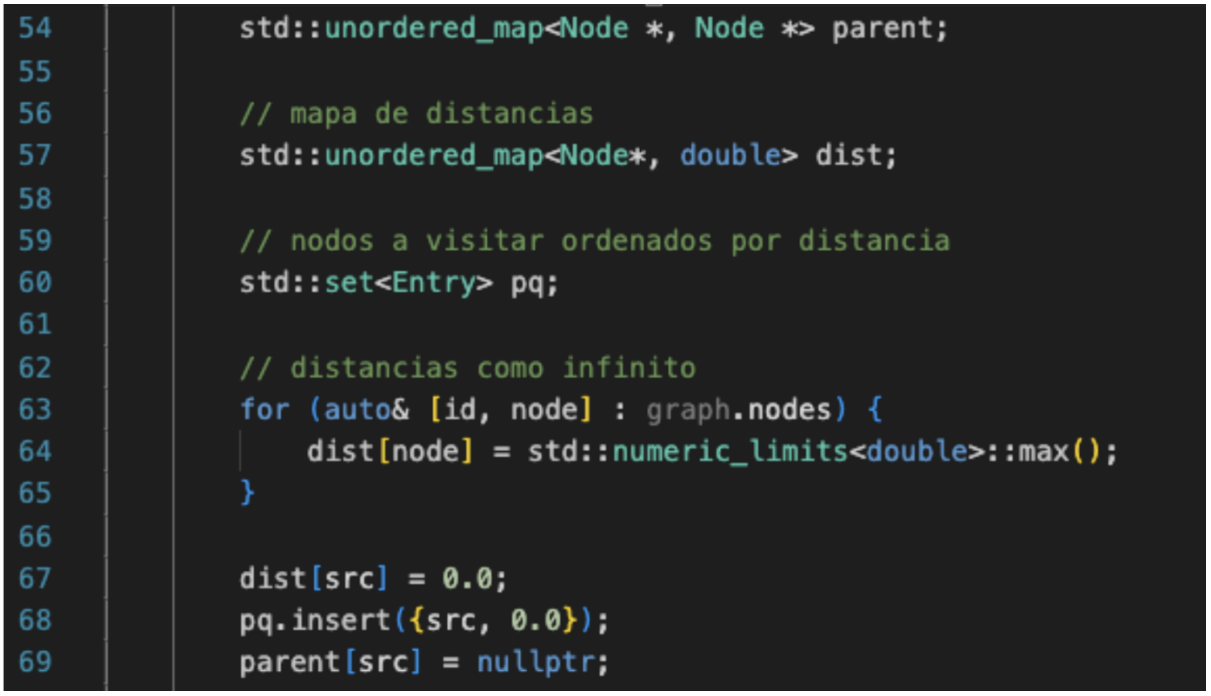
Análisis de Complejidad Algorítmica y Espacial para **Dijkstra:**



```
53    void dijkstra(Graph &graph) {
54        std::unordered_map<Node *, Node *> parent;
55
56        // mapa de distancias
57        std::unordered_map<Node*, double> dist;
58
59        // nodos a visitar ordenados por distancia
60        std::set<Entry> pq;
61
62        // distancias como infinito
63        for (auto& [id, node] : graph.nodes) {
64            dist[node] = std::numeric_limits<double>::max();
65        }
66
67        dist[src] = 0.0;
68        pq.insert({src, 0.0});
69        parent[src] = nullptr;
70
71        int iterations = 0;
72        while (!pq.empty()) {
73            Entry current_entry = *pq.begin();
74            pq.erase(pq.begin());
75            Node* current = current_entry.node;
76
77            iterations++;
78
79            /*
80            if (iterations % 1000 == 0) {
81                std::cout << "Dijkstra iteration " << iterations << ", queue size: " << pq.size() << std::endl;
82            }
83            */
84
85            if (current == dest) {
86                std::cout << "Dijkstra llego al destino despues de " << iterations << " iteraciones" << std::endl;
87                break;
88            }
89
90            // explorar todas las aristas del nodo actual
91            for (Edge* edge : current->edges) {
92                Node* neighbor = nullptr;
93                if (edge->src == current) {
94                    neighbor = edge->dest;
95                } else if (!edge->one_way && edge->dest == current) {
96                    neighbor = edge->src;
97                }
98
99                if (neighbor == nullptr) continue;
100
101               // nueva distancia al vecino
102               double new_dist = dist[current] + edge->length;
103
104               // si hay un camino mas corto
105               if (new_dist < dist[neighbor]) {
106                   // se quita la entrada anterior del set
107                   pq.erase({neighbor, dist[neighbor]});
108
109                   dist[neighbor] = new_dist;
110                   parent[neighbor] = current;
111
112                   pq.insert({neighbor, new_dist});
113
114                   visited_edges.push_back(sfLine(
115                       current->coord,
116                       neighbor->coord,
117                       sf::Color(100, 100, 255, 100),
118                       1.0f
119                   ));
120
121                   render();
122               }
123           }
124       }
125
126       set_final_path(parent);
127   }
```

Complejidad computacional

En el análisis no se considera el costo de render ni de las estructuras de visualización. Al inicio, el recorrido de graph.nodes para inicializar dist en infinito es O(V) y la creación de parent, dist, pq y closed_set añade también O(V) en tiempo y espacio.



```
54      std::unordered_map<Node *, Node *> parent;
55
56      // mapa de distancias
57      std::unordered_map<Node*, double> dist;
58
59      // nodos a visitar ordenados por distancia
60      std::set<Entry> pq;
61
62      // distancias como infinito
63      for (auto& [id, node] : graph.nodes) {
64          dist[node] = std::numeric_limits<double>::max();
65      }
66
67      dist[src] = 0.0;
68      pq.insert({src, 0.0});
69      parent[src] = nullptr;
```

El núcleo del algoritmo está en el while (!pq.empty()). En cada iteración, lo primero que se hace es extraer de la cola de prioridad el nodo con menor distancia acumulada. Esta operación (top + pop sobre el min-heap) tiene un costo O(log V), donde V es el número de nodos. A lo largo de toda la ejecución, puede haber hasta O(V) extracciones, lo que aporta un costo total de O(V log V).

```
71      int iterations = 0;
72      while (!pq.empty()) {
73          Entry current_entry = *pq.begin();
74          pq.erase(pq.begin());
75          Node* current = current_entry.node;
76
77          iterations++;
78
79          /*
80          if (iterations % 1000 == 0) {
81              std::cout << "Dijkstra iteration " << iterations << ", queue size: " << pq.size() << std::endl;
82          }
83          */
84
85          if (current == dest) {
86              std::cout << "Dijkstra llego al destino despues de " << iterations << " iteraciones" << std::endl;
87              break;
88          }
```

Luego, para ese nodo extraído, se recorren todas sus aristas incidentes mediante el for (Edge* edge : current->edges). Si el nodo actual tiene grado d, el costo de esta parte en esa iteración es O(d). Sumando sobre todas las iteraciones del bucle, la suma de todos los grados es del orden de E (cada arista se visita un número constante de veces), por lo que el costo total de recorrer las aristas es O(E).

```
90              // explorar todas las aristas del nodo actual
91              for (Edge* edge : current->edges) {
92                  Node* neighbor = nullptr;
93                  if (edge->src == current) {
94                      neighbor = edge->dest;
95                  } else if (!edge->one_way && edge->dest == current) {
96                      neighbor = edge->src;
97                  }
98
99                  if (neighbor == nullptr) continue;
100
101                 // nueva distancia al vecino
102                 double new_dist = dist[current] + edge->length;
103
```

Dentro de ese recorrido de aristas, cada vez que se encuentra un camino más corto hacia un vecino, se actualizan las tablas hash dist y parent (operaciones promedio O(1)) y se inserta el vecino en la cola de prioridad con un push, que cuesta O(log V). En el peor caso, cada arista puede producir una de estas inserciones, de modo que puede haber hasta O(E) operaciones de push, con un costo total O(E log V).

```
103
104                 // si hay un camino mas corto
105                 if (new_dist < dist[neighbor]) {
106                     // se quita la entrada anterior del set
107                     pq.erase({neighbor, dist[neighbor]});
108
109                     dist[neighbor] = new_dist;
110                     parent[neighbor] = current;
111
112                     pq.insert({neighbor, new_dist});
113
114                     visited_edges.push_back(sfLine(
115                         current->coord,
116                         neighbor->coord,
117                         sf::Color(100, 100, 255, 100),
118                         1.0f
119                     ));
120
121                     render();
122                 }
123             }
124         }
125
126         set_final_path(parent);
127     }
```

Combinando estas partes del bucle (extracciones O(V log V), recorrido de aristas O(E) e inserciones O(E log V)), la complejidad global del algoritmo es O((V + E) log V), que habitualmente se expresa como O(E log V). En grafos dispersos (E ≈ O(V)), esto se aproxima a **O(V log V)**; en grafos densos (E ≈ V^2), se acerca a **O(V^2 log V)**. La condición de parada cuando se llega a dest puede mejorar el tiempo en casos prácticos, pero no modifica la cota de peor caso.

Complejidad espacial

No se considera el espacio del grafo ni de las estructuras de dibujo, solo la memoria extra del algoritmo.

- dist: guarda una distancia por cada nodo = O(V).
- parent: guarda el padre de cada nodo = O(V).
- closed_set: puede llegar a contener todos los nodos procesados = O(V).

- pq (priority queue): en el peor caso puede almacenar muchos nodos pendientes, pero su tamaño relevante también se acota por el número de nodos = O(V).

Sumando todo, el espacio adicional que necesita Dijkstra, aparte del grafo, es del orden de **O(V).**

Análisis de Complejidad Algorítmica y Espacial para **Best First Search:**

```cpp
void best_first_search(Graph &graph) {
    std::unordered_map<Node *, Node *> parent;

    // Set de nodos a visitar ordenados por heuristica
    std::set<Entry> open_set;

    // conjunto de nodos visitados
    std::unordered_set<Node*> visited;

    auto heuristic = [this](Node* node) -> double {
        float dx = node->coord.x - dest->coord.x;
        float dy = node->coord.y - dest->coord.y;
        return std::sqrt(dx * dx + dy * dy);
    };

    // inicializar el nodo origen
    open_set.insert({src, heuristic(src)});
    parent[src] = nullptr;

    // mientras haya nodos por visitar
    int iterations = 0;
    while (!open_set.empty()) {
        // nodo con menor euristica
        Entry current_entry = *open_set.begin();
        open_set.erase(open_set.begin());
        Node* current = current_entry.node;

        // marcar como visitado
        visited.insert(current);

        iterations++;
        /*
        if (iterations % 1000 == 0) {
            std::cout << "Best-First Search iteration " << iterations << ", open set size: " << open_set.size() << std::endl;
        }
        */

        // si se llega al destino, break
        if (current == dest) {
            std::cout << "Best-First Search llego a su destino despues de " << iterations << " iteraciones" << std::endl;
            break;
        }

        // se exploran todas las aristas del nodo actual
        for (Edge* edge : current->edges) {
            // determinar el nodo vecino segun la direccion de la arista
            Node* neighbor = nullptr;
            if (edge->src == current) {
                neighbor = edge->dest;
            } else if (!edge->one_way && edge->dest == current) {
                neighbor = edge->src;
            }

            if (neighbor == nullptr) continue;

            // si el vecino no ha sido visitado.
            if (visited.find(neighbor) == visited.end()) {
                // calcular heuristica del vecino
                double h = heuristic(neighbor);

                // actualizar el padre
                parent[neighbor] = current;
```

```
306                 // actualizar el padre
307                 parent[neighbor] = current;
308
309                 // insertar en el set de los abiertos
310                 open_set.insert({neighbor, h});
311
312                 // marcar en los visitados
313                 visited.insert(neighbor);
314
315                 visited_edges.push_back(sfLine(
316                     current->coord,
317                     neighbor->coord,
318                     sf::Color(255, 100, 255, 100),  // magenta
319                     1.0f
320                 ));
321
322                 render();
323             }
324         }
325     }
326
327     set_final_path(parent);
328 }
```

Para realizar el análisis algorítmico, este se va a realizar por partes (el render no es parte del análisis, ni el dibujo); en el cual:

- V: número de vértices
- E: número de aristas

La inicialización de variables tiene complejidad O(1), al igual que la función heurística. El insert en un set (árbol balanceado) es de O(log V), sin embargo, al inicio solo hay 1 elemento, por lo que es O(1).

```
std::unordered_map<Node *, Node *> parent;

// Set de nodos a visitar ordenados por heuristica
std::set<Entry> open_set;

// conjunto de nodos visitados
std::unordered_set<Node*> visited;

auto heuristic = [this](Node* node) -> double {
    float dx = node->coord.x - dest->coord.x;
    float dy = node->coord.y - dest->coord.y;
    return std::sqrt(dx * dx + dy * dy);
};

// inicializar el nodo origen
open_set.insert({src, heuristic(src)});
parent[src] = nullptr;
```

Sobre el bucle principal: Tomar el primer elemento del set tiene complejidad O(1), erase sobre el árbol balanceado (open set) en el peor de los casos va a ser O(log V). Asimismo, la inserción sobre el unordered_set de visitados tiene complejidad O(1) amortiguado. (La extracción del mejor nodo cuesta O(log V).

```
int iterations = 0;
while (!open_set.empty()) {
    // nodo con menor euristica
    Entry current_entry = *open_set.begin();
    open_set.erase(open_set.begin());
    Node* current = current_entry.node;

    // marcar como visitado
    visited.insert(current);
```

Sobre el for interior para el recorrido de vecinos; por cada arista que sale de current, encontrar el vecino es O(1) amortiguado, calcular la heurística al igual es O(1), actualizar parent es O(1) amortiguado, la inserción sobre el open set es O(log V) y la inserción sobre visitados es O(1) amortiguado.

Nota: Cuando se refiere amortiguado, es porque en los unordered_map, es posible que sucedan operaciones de re-hash.

```
for (Edge* edge : current->edges) {
    // determinar el nodo vecino segun la direccion de la aris
    Node* neighbor = nullptr;
    if (edge->src == current) {
        neighbor = edge->dest;
    } else if (!edge->one_way && edge->dest == current) {
        neighbor = edge->src;
    }

    if (neighbor == nullptr) continue;

    // si el vecino no ha sido visitado.
    if (visited.find(neighbor) == visited.end()) {
        // calcular heuristica del vecino
        double h = heuristic(neighbor);

        // actualizar el padre
        parent[neighbor] = current;

        // insertar en el set de los abiertos
        open_set.insert({neighbor, h});

        // marcar en los visitados
        visited.insert(neighbor);

        visited_edges.push_back(sfLine(
            current->coord,
            neighbor->coord,
            sf::Color(255, 100, 255, 100),  // magenta
            1.0f
        ));
```

Cálculos finales: En esta implementación se marca visited cuando se realiza la inserción sobre el open_set, no cuando es sacado. Lo que significa que cada nodo se inserta a lo sumo una vez en open_set. Por lo que cada nodo se inserta en open_set una vez y se extrae de open_set una vez.

Sobre la iteraciones del open_set: El costo es O(V log V) por Las V inserciones + V borrados y cada uno cuesta log V.

Visited: Cada arista se procesa a lo sumo un número constante de veces: O(E).

Complejidad Computacional Total: O(V log V) + O(E) = O(E + V log V).

Análisis espacial:

1. unordered_map parent: Este guarda como máximo un padre por cada nodo alcanzado, lo cual es O(V).
2. unordered_set visited: Este marca los nodos descubiertos o visitados, en el peor caso se marca todos los nodos del grafo, por lo que es O(V).
3. set open_set: Contiene los nodos aún por explorar, en el peor caso puede llegar a tener una gran parte de los nodos, por lo que es O(V).
4. Si en el caso tomaramos en cuenta visited_edges (para el dibujo), este en el peor de los casos podría almacenar una cantidad proporcional a las aristas exploradas, por lo que es O(E).

Complejidad espacial: O(V)