Ramificación y poda

Práctica Final
26 de mayo de 2025

Juan Llinares Mauri
74011239E

Análisis y Diseño de Algoritmos

Ingeniería Informática
Universidad de Alicante
España

En esta memoria se pretende documentar el desarrollo de la práctica final de Análisis y Diseño de Algoritmos[1]. El problema a resolver será una batería de laberintos diferentes y la metodología a aplicar será ramificación y poda.

Índice

[1]El código plasmado en la memoria no tiene por qué ser igual al que haya sido entregado. En la memoria se pretende documentar el desarrollo del código y sus respectivas mejoras a lo largo de la realización de la práctica.

1. Estructuras de datos.

Deberemos usar una estructura organizada para poder comparar las diferentes características de los nodos y poder ordenarlos de mejor a peor en la cola de prioridad.

1.1. Nodo.

Para representar un nodo en nuestro problema crearemos la siguiente estructura:

```
struct Node
{
    int x = 0, y = 0;
    vector<Step> path;
    int cost = 0;
};
```

El nodo consta de dos variables enteras que representan la posición del mismo, el coste del propio nodo y el camino codificado que se ha seguido para llegar hasta él. Este camino es un vector de otra `struct` personalizada llamada `Step`. Esta estructura representa todas las direcciones posibles y está declarada en el código como:

```
enum Step
{
    N, NE, E, SE, S, SW, W, NW
};
```

1.2. Lista de nodos vivos.

La lista de nodos vivos será una *priority queue* con la siguiente estructura:

```
priority_queue<Node, vector<Node>, compareNodes> q;
```

Esta cola de prioridad se instancia dentro del algoritmo principal. El método `maze_bb` contiene el algoritmo.

La estructura `compareNodes` comparará dos nodos dados para poder elegir el de menor coste. Contiene la siguiente lógica:

```
struct compareNodes
{
    bool operator()(const Node &a, const Node &b)
    {
        return a.cost > b.cost;
    }
};
```

La función usa el operador `>` en estructuras personalizadas. Se ha sobrecargado este operador para que pueda comparar dos nodos dados. Contiene la siguiente lógica:

```
bool operator<(const Node &a, const Node &b)
{
    return (a.cost + optimist(a)) > (b.cost + optimist(b));
}
```

La evaluación se realiza teniendoo en cuenta el coste que tenga cada nodo añadiéndole a este la cota optimista asociada a ese mismo nodo. A mayor valor de la suma, mayor posición en la lista de prioridad tendrá el nodo y viceversa.

Las implementación de las cotas optimistas se pueden ver en la sección 3.3.

1.3. Laberinto

Se ha creado una estructura personalizada para poder guardar el laberinto y que sea accesible desde cualquier contexto:

```
struct Maze
{
    int n = 0;
    int m = 0;
    vector<vector<int>> maze = {};
};
```

Con esta estructura en el código, se ha creado una variable global que almacena el laberinto llamada `mazeStruct`. Esta variable será la que usen todas las funciones que necesiten acceder al laberinto. Se ha hecho de esta manera para evitar el paso del laberinto como de parámetro a todas las funciones y por claridad en el código.

1.4. Estadísticas

Para poder llevar un recuento de las estadísticas de manera organizada se ha creado otra `struct` personalizada llamada `Stats` que cuenta con los siguientes campos:

```
struct Stats
{
    int nv = 0;                     // Numero de nodos visitados
    int ne = 0;                     // Numero de nodos explorados
    int nhv = 0;                    // Numero de nodos hoja visitados
    int nnf = 0;                    // Numero de nodos no factibles
    int nnp = 0;                    // Numero de nodos no prometedores
    int ppd = 0;                    // Numero de nodos prometedores pero
        descartados
    int msah = 0;                   // Numero de veces que la mejor
        solucion se ha actualizado desde un nodo hoja
    int msap = 0;                   // Numero de veces que la mejor
        solucion se ha actualizado a partir de la cota pesimista de un nodo
         sin completar
    std::chrono::duration<double> t; // Tiempo de ejecucion
};
```

Cuenta con las estadísticas sobre los nodos necesarias y el tiempo de ejecución.

2. Mecanismos de poda.

Es importante instanciar de manera correcta los mecanismos de poda que seguiremos para la resolución de este problema.

2.1. Poda de nodos no factibles.

Los nodos que no sean factibles serán evaluados con el siguiente método:

```
bool isFeasible(const Node &node)
{
    return node.x >= 0 && node.x < mazeStruct.n &&
           node.y >= 0 && node.y < mazeStruct.m &&
           mazeStruct.maze[node.x][node.y] == 1;
}
```

A este método se le llama dentro del bucle de expansión del nodo actual. Se puede ver el esquema de la implementación seguida en la sección 2.3.

2.2. Poda de nodos no prometedores.

Los nodos no prometedores se evaluarán mediante la siguiente función:

```
bool isPromising(const Node &node, const Node &currentBest)
{
    return (node.cost + optimist(node)) < currentBest.cost;
}
```

La función `optimist` se verá en la sección 3.3. Esta función que comprueba si un nodo es prometedor es llamada dos veces:

- Cuando se elige el primer nodo de la cola de prioridad para comprobar si puede ser descartado al ser no prometedor (línea 14 del esquema de la sección 2.3).
- Después de comprobar que un nodo expandido es factible para comprobar si es prometedor (línea 34 del esquema de la sección 2.3).

2.3. Esquema de implementación

El esquema que se ha seguido para implementar el código de esta práctica ha sido el siguiente:

```
void maze_bb(Node &currentBest, Stats &stats, bool debug)
{
    Node initial = initNode();

    //...

    if (mazeStruct.maze[0][0] == 0)
        // ...

    while (!q.empty()) {
        Node n = q.top();
        q.pop();

        if (!isPromising(n, currentBest))
            // ...

        if (isLeaf(n)) {
            // ...

            if (isBetter(solution(n), solution(currentBest)))
                // ...
            continue;
        }

        for (Node a : expand(n)) {
            if (isFeasible(a)) {
                // ...

                if (isBetter(solution(pes(a, debug)), solution(currentBest)
                    ))
                    // ...

                if (isPromising(a, currentBest)) {
                    q.push(a);
                    // ...
                }
                else
                    // ...
            }
            else
                // ...
        }
    }
}
```

3. Cotas pesimistas y optimistas.

En esta sección de la memoria veremos las tres cotas que usaremos durante el algoritmo.

3.1. Cota pesimista inicial (inicialización).

Para inicializar los nodos se usa la siguiente función:

```
Node initNode()
{
    Node initial = Node();
    initial.x = 0;
    initial.y = 0;
    initial.cost = 0;
    initial.path = vector<Step>();

    return initial;
}
```

Dentro de `maze_bb()` llamamos a esta función `initNode()`. Justo después de ella, llamamos a otra función llamada `pessimist()`. Este último método se encargará de establecer la cota pesimista para un nodo dado siendo, en este caso, el nodo inicial.

La función `pessimist()` es la siguiente:

```
Node pessimist(const Node &node, bool debug)
{
    Node pesimista = node;
    pesimista.cost = INT_MAX;

    Node bfsNode = bfs();

    if (bfsNode.cost >= 0)
        pesimista = bfsNode;

    return pesimista;
}
```

Se establece un valor `INT_MAX` por si el algoritmo pesimista no encuentra solución.

El algoritmo escogido para la poda pesimista es una búsqueda en anchura. Antes de escoger esta opción, se probó con el algoritmo *Greedy* de la práctica 7, pero obtenemos mejores resultados con la búsqueda en anchura (ver sección 6).

3.2. Cota pesimista del resto de nodos.

La cota pesimista es la que usaremos para comprobar si un nodo puede ser podado, es decir, si podemos descartar un estado de la resolución del laberinto sin explorarlo más a fondo porque se sabe que no conducirá a una solución óptima.

Para todos los nodos usamos el resultado que ofrece la búsqueda en anchura. El esquema de este algoritmo es el siguiente:

```
Node bfs()
{
    // ...
    vector<vector<bool>> seen(N, vector<bool>(M, false));
    vector<vector<Step>> parentStep(N, vector<Step>(M));
    vector<vector<pair<int, int>>> parentPos(N, vector<pair<int, int>>(M));
    queue<tuple<int, int, int>> q;

    if (mazeStruct.maze[0][0] == 0)
        // Devolvemos INT_MAX
```

```
11
12      q.push({0, 0, 1});
13      seen[0][0] = true;
14
15      // ...
16
17      while (!q.empty()) {
18          auto [i, j, d] = q.front();
19          q.pop();
20
21          if (i == N - 1 && j == M - 1) { // Si final del laberinto...
22              // ...
23
24              while (!(ci == 0 && cj == 0))
25                  // Reconstruimos el camino
26
27              // Creamos el nodo resultado y lo devolvemos
28
29              return result;
30          }
31
32          for (int k = 0; k < 8; ++k) { // Expandimos el nodo
33              // ...
34
35              if (/* Es factible y no visitado antes */) {
36                  // Pusheamos la nueva posicion
37              }
38          }
39      }
40
41      // Devolvemos INT_MAX
42  }
```

Con este algoritmo obtenemos una solución rápida y factible para realizar una poda.

Es una buena cota pesimista porque su coste nunca será más de $O(n * m)$ y siempre nos garantiza una solución de manera eficiente.

3.3. Cota optimista.

La cota optimista es la mejor solución que se le puede dar al problema del laberinto.

La estrategia que se ha elegido para esta práctica se puede ver reflejada en el código en la función llamada `optimist`:

```
1  int optimist(const Node &node)
2  {
3      // return chebyshev(node);
4      // return euclidean(node);
5      return dist[node.x][node.y];
6  }
```

Como se puede ver en los comentarios de dentro de la función, se ha probado con varias metodologías:

- **Distancia de Chebyshev:** Distancia desde la celda actual al final del laberinto usando la heurística `max(abs(node.x - (mazeStruct.n - 1)), abs(node.y - (mazeStruct.m - 1)))`.
- **Distancia de Manhattan o Euclídea:** Distancia desde la celda actual al final del laberinto usando la heurística `sqrt(pow(node.x - (mazeStruct.n - 1), 2) + pow(node.y - (mazeStruct.m - 1), 2))`.
- **Distancia al final desde cada celda:** Distancia desde cada celda al final del laberinto. Se usa una función llamada `precomputeDistances` que guarda la distancia mínima desde cada

celda camino al final del laberinto. Estas distancias se guardan en un `vector` de `int` doble llamado `dist`.

4. Otros medios empleados para acelerar la búsqueda.

Los métodos usados para mejorar el rendimiento del algoritmo se han comentado a lo largo de la memoria hasta ahora.

Cabe destacar que el algoritmo de búsqueda en anchura obtiene el resultado y el camino correcto en tiempos muy bajos para todos los laberintos que tenemos disponibles en la batería de ficheros de pruebas (ver sección 6). Esto se debe a que la complejidad en el peor caso de esta heurística se acota por $O(n * m)$ siendo n y m el tamaño del laberinto. Usando ramificación y poda, el orden es exponencial acorde con el número de nodos explorados aunque se realicen podas en la mayoría de ellos.

4.1. Expansión de hijos

Para acelerar la búsqueda, se ordenan los hijos `node` cuando se expande el nodo padre para que el primero en ser explorado sea el que menos `node.cost + chebyshev(node)` tenga. En el código, este aspecto se ve en la función `expand`, concretamente al final y está programado de la siguiente manera:

```
1 std::sort(children.begin(), children.end(), [n, m](const Node &a, const
      Node &b) {
2     int fA = a.cost + chebyshev(a);
3     int fB = b.cost + chebyshev(b);
4     return fA < fB; });
5
6 return children;
```

5. Estudio comparativo de distintas estrategias de búsqueda.

En esta sección de la memoria veremos los tiempos que se obtienen para los laberintos de esta práctica usando *backtracking*. No podemos usar las anteriores estrategias a *backtracking* porque sólo contaban con tres tipos de movimientos diferentes (abajo, derecha y abajo-derecha) y la mayoría de los laberintos de esta práctica no tendrían solución usando esos algoritmos.

Nº laberinto	*Backtracking* (s)	Ramificación y poda (s)
0	0.000	0.000
1	0.000	0.000
2	0.000	0.000
3	0.000	0.000
4	0.000	0.000
5	0.000	0.000
6	0.000	0.000
7	0.008	0.000
8	0.000	0.002
9	0.000	0.000
10	?	0.028
11	0.002	0.006
12	0.002	0.000
13	0.668	?
...	?	?

Se puede observar que los dos algoritmos desarrollados están a la par en cuanto a tiempo, aunque ninguno sabe resolver a partir del laberinto 13.

6. Tiempos de ejecución.

En esta sección se recogen los tiempos para todos los laberintos que han sido resueltos por ramificación y poda.

Fichero de test	Tiempo (ms)
00-bb.maze	0.000
01-bb.maze	0.011
02-bb.maze	0.098
03-bb.maze	0.020
04-bb.maze	0.027
05-bb.maze	0.139
06-bb.maze	0.012
07-bb.maze	0.219
08-bb.maze	1.866
09-bb.maze	0.491
10-bb.maze	28.480
11-bb.maze	7.351
12-bb.maze	0.368

6.1. Tiempos de BFS

Sección extra para indicar los tiempos que obtenemos mediante BFS (*Breath-First Search*) o búsqueda en anchura. Para esta sección se ha extraído el método `Node bfs()` en un fichero aparte, se ha compilado y se ha probado de manera independiente. Sólo recogeremos algunos de los laberintos más grandes de la fase de pruebas.

Fichero de test	Tiempo (ms)
k01-bb.maze	56.687
k02-bb.maze	76.130
k03-bb.maze	519.598
k05-bb.maze	1537.237
k10-bb.maze	3078.001