Esteban Antón Sellés - 48788593G

Índice

1. Estructura de datos

INTRODUCCIÓN DE DATOS DE IMPORTANCIA

1.1. Nodo

En esta práctica, el nodo se ha creado a partir de la definición de un tipo de dato `tuple` llamado `Node`.

```
typedef tuple<unsigned long, pair<short, short>, pair<short, short>, unsigned long> Node;
```

Para que a la hora de implementar el código hubiese una mayor claridad a la hora de utilizar este dato, se ha creado un `Enum` con nombres que hacen referencia al uso de cada componente del nodo. Para una mejor explicación se mostrará este y la explicación de cada una de sus componentes, que se referieren a cada dato de la tupla respectivamente.

```
// Para pedir los datos de un nodo
enum Node_values
{
  STEPS,    // Pasos recorridos desde la primera casilla hasta su posición
  POS,   // Posición actual
  PREV,   // Posición previa a esta (de dónde vengo)
  OPTIM   // Cota optimista
};
```

- STEPS (unsigned long): Almacena los “pasos” o movimientos que se han llevado a cabo desde la posición inicial, (0, 0), hasta la posición actual a la que el nodo representa.
- POS (pair<short, short>): Almacena la posición del laberinto a la que el nodo representa
- PREV (pair<short, short>): Almacena la posición previa de la que viene. Se podría decir que se acuerda de quién es su padre, por lo que se puede saber de dónde viene.
- OPTIM (unsigned long): Almacena la cota optimista de dicho nodo.

1.2. Lista de nodos vivos

Como se ha visto en clase de teoría y según se explica en el material de la asignatura, al tratarse de una búsqueda ordenada de los nodos más prometedores, se trata de una búsqueda dirigida. Esto sirve de justificación a la hora de haber elegido una cola de prioridad para la tarea de almacenar los nodos vivos, o los nodos que en el momento de la expansión de su predecesor han sido valorados como prometedores y que más tarde serán comprobados de nuevo para su posterior expansión.

Esta cola de prioridad se utiliza desde la función `branchAndBoundAlgorithm()` de la siguiente forma:

- Declaración e inicialización:
  Se inicializa con `initial_node`, el cual es el primer nodo creado, el equivalente a haber dado el primer paso en el laberinto en a la casilla (0,0).

```
// Cola de prioridad
priority_queue<Node> pq;
pq.push (initial_node);
```

- Extracción del nodo más prometedor:
  Para esto se ha sobrecargado el operador < de la siguiente forma:

```
// Sobrecarga de operador para que la cola se ordene por orden de mejor cota
bool operator<(const Node &a, const Node &b)
{
  return get<OPTIM>(a) > get<OPTIM>(b);
}
```

Como se puede observar, la obtención del nodo más prometedor se consigue con la comparación de la cota optimista de cada nodo. El nodo que mayor cota optimista posea, será considerado el nodo más prometedor de la cola de prioridad.

La necesidad de la sobrecarga de este operador es la de querer cambiar el comportamiento de la función `top()` de la librería `<queue>`, la cual devuelve el elemento más prioritario de la cola.

- Función `pop()`:
  Esta función elimina o extrae el primer elemento de la lista, con esto se asegura que el nodo devuelto por la función `top()` desaparezca de la lista.

```
pq.pop();
```

2. Mecanismos de poda

2.1. Poda de nodos no factible

Un nodo se considera no factible por tres razones:

- Su posición se excede de los límites del laberinto
- El valor de la posición es 0
- Los pasos que ha recorrido para llegar a esa posición son mayores que los que ha tenido que recorrer otro nodo para alcanzar la misma posición

```
// Es posible, está en un lugar en el que se puede estar
bool is_feasible(const Node &node)
{
  bool feasible = true;

  if (get<POS>(node).first < 0 || get<POS>(node).second < 0 ||
      get<POS>(node).first >= n || get<POS>(node).second >= m)
  {
    feasible = false;
    no_feasible++;
  }
  else if (maze[get<POS>(node).first][get<POS>(node).second] == 0 ||
           get<STEPS>(node) >= warehouse[get<POS>(node).first][get<POS>(node).second])
  {
    feasible = false;
    no_feasible++;
  }

  return feasible;
}
```

2.2. Poda de nodos no prometedores

Para la poda de los nodos no prometedores, primero un nodo debe ser factible. Para que un nodo sea prometedor debe de tener una cota optimista menor o igual a la cota pesimista.

```
// Es prometedor, da buenas cotas
bool is_promising(const Node &node)
{
  bool promising = true;

  if (get<OPTIM>(node) > solution)
  {   // No es un buen nodo para expandir por sus cotas
    promising = false;
    no_promising++;
  }
  warehouse[get<POS>(node).first][get<POS>(node).second] = get<STEPS>(node);

  return promising;
}
```

El uso de estas dos podas se ha llevado a cabo en el interior de la función `branchAndBoundAlgorithm()`, con lo que si el nodo pasaba las dos podas se insertaba en la cola de prioridad de nodos vivos, para su posterior expansión.

```
// En cada llamada es un nuevo nodo, con lo que vuleve a estudiar todo de nuevo
unsigned long branchAndBoundAlgorithm()
{
  explored++;
  // Inicialización de valores
  warehouse = vector< vector<unsigned long> >(n, vector<unsigned long>(m, numeric_limits<unsigned long>::max()));
  Node initial_node(solution++, make_pair(0, 0), make_pair(0, 0), optimistic_height(solution, 0, 0));
  // Las cotas
  optimistic_value = optimistic_height(get<STEPS>(initial_node), get<POS>(initial_node).first, get<POS>(initial_node).second);
  solution = pessimistic_height(initial_node);
  // Cola de prioridad
  priority_queue<Node> pq;
  pq.push (initial_node);
  // Se actualiza memoización
  warehouse[get<POS>(initial_node).first][get<POS>(initial_node).second] = get<STEPS>(initial_node);

  while (!pq.empty())
  {
    // Se extrae el nodo más prometedor
    Node aux_n = pq.top();
    pq.pop();

    if (is_leaf(aux_n))
    {   // Caso base - Ser una hoja significa el fin del laberinto
      if (is_better(aux_n))
      {
        solution = get<STEPS>(aux_n);
      }
    }
    else
    {   // Sigue expandiendo
      for (Node a : expand(aux_n))
      {
        if (is_feasible(a))
        {   // Si es posible
          warehouse[get<POS>(a).first][get<POS>(a).second] = get<STEPS>(a);
          optimistic_value = optimistic_height(solution, get<POS>(a).first, get<POS>(a).second);
          if (is_promising(a))
          {
            pq.push(a);
            added++;
          }
        }
      }
    }
  }

  return solution;
}
```

3. Cotas pesimistas y optimistas

```
unsigned long optimistic_height(const unsigned long &sol, const short &x, const short &y)
{
  unsigned long optimistic = 0;

  optimistic = sol + sqrt(pow(n - x, 2) + pow(m - y, 2));

  return optimistic;
}

unsigned long greedyAlgorithm(const Node &node, bool &blocked)
{
  unsigned long path = 0;
  short i = get<POS>(node).first;
  short j = get<POS>(node).second;
  int move;
  bool end = false;

  // Si la posiciión (0, 0) es 0 es imposible hacer el laberinto
  if (maze[i][j] == 0)
  {
    blocked = true;
  }

  while (!blocked && (i < n || j < m) && !end)
  {
    path++;
    move = calculate(i, j);
    if (move == 0)
    {   // Diagonal
      i++;
      j++;
    }
    else if (move == 1)
    {   // Derecha
      j++;
    }
    else if (move == -1)
    {   // Abajo
      i++;
    }
    else if (i == n-1 && j == m-1)
    {
      end = true;
    }
    else
    {
      blocked = true;
    }
  }
  return path;
}
// ########################################################################

// Solución voraz
unsigned long pessimistic_height(const Node &n)
{
  unsigned long pessimistic = 0;
  bool blocked = false;

  pessimistic = greedyAlgorithm(n, blocked);

  if (blocked)
  {
    pessimistic = numeric_limits<unsigned long>::max();
  }

  return pessimistic;
}
```

3.1. Cota pesimista y optimista del nodo inicial

Al comenzar a resolver el problema, la cota optimista del nodo inicial es el camino más corto entre la posición inicial y la casilla final.
Por el contrario, la cota pesimista se estudia con la solución voraz desde el punto inicial.

3.2. Cota pesimista y optimista de los demás nodos

Como en el nodo inicial, la cota optimista se calcula con el camino más corto entre la posición del nodo y la salida, sin tener en cuenta si esa solución es real o no. En cuanto a la cota pesimista, se calcula con la solución voraz desde el nodo a la salida y se le suma el camino recorrido hasta entonces para encontrarse en esa posición.

4. Otros medios empleados para acelerar la búsqueda

NO IMPLEMENTADO

5. Estudios comparativos de distintas estrategias de búsqueda

NO IMPLEMENTADO

6. Soluciones y tiempos de ejecución

- Fichero 20.8maze: 0.581 ms
- Fichero 21.8maze: 0.032 ms
- Fichero 22.8maze: 0.042 ms
- Fichero 23.8maze: 2.039 ms
- Fichero 24.8maze: ¿?
- Fichero 25.8maze: 1.735 ms
- Fichero 26.8maze: 2.909 ms
- Fichero 27.8maze: 4.161 ms
- Fichero 28.8maze: 15.261 ms
- Fichero 29.8maze: ¿?
- Fichero 30.8maze: ¿?