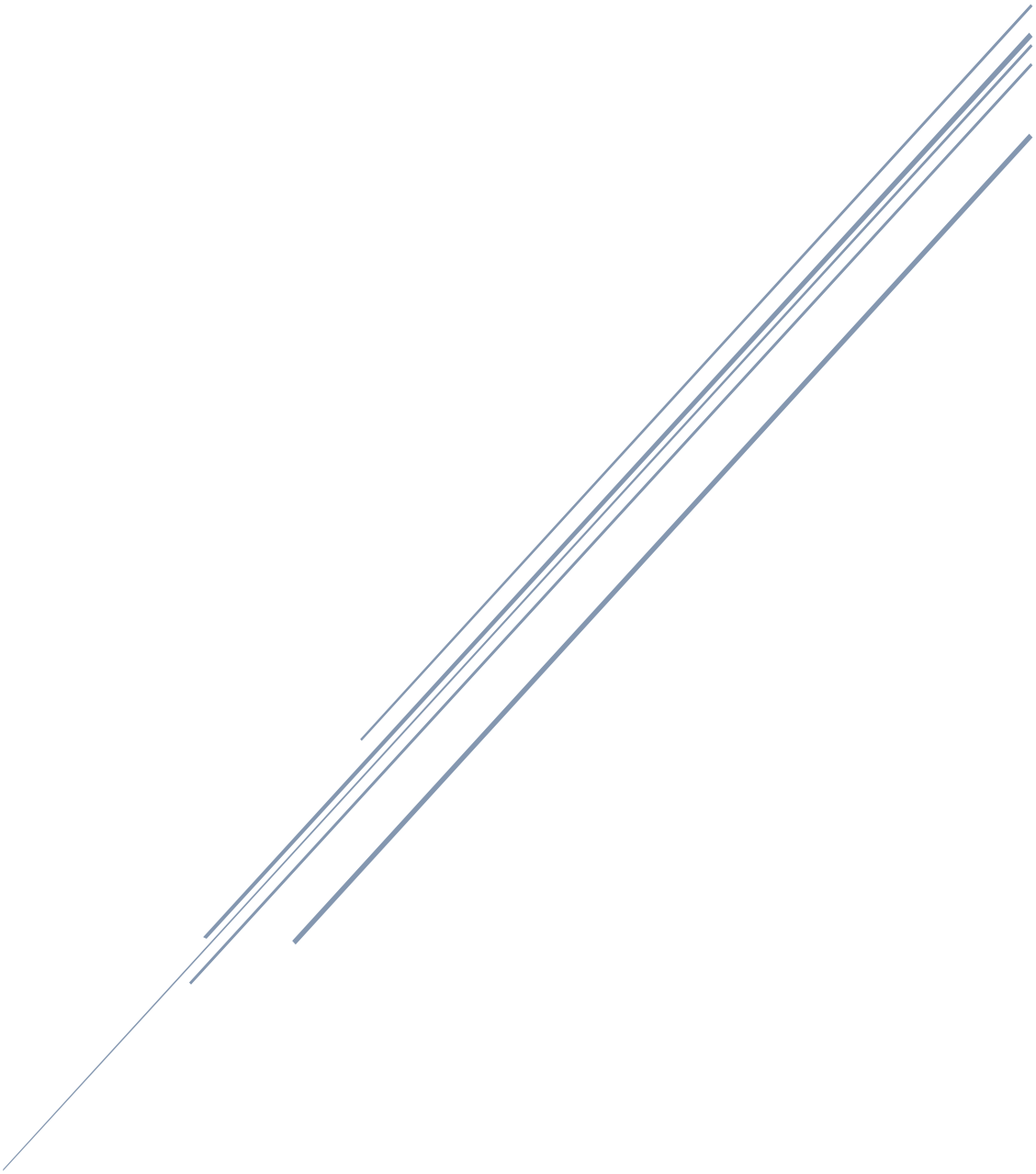
PRÁCTICA FINAL

Análisis y diseño de algoritmos.



Adam Roy Frederick William Reading
Y2483358Q

Contenido

Estructura de datos

Nodo

En cada nodo se almacena la fila y la columna donde se encuentra, la cota optimista para ese nodo, el camino tomado para llegar a ese nodo y finalmente un contador que indicará cuántos movimientos se ha realizado desde el inicio hasta ese nodo.

```
struct Node{
    vector<int> path;
    int fila;
    int columna;
    int optimista;
    int k;
};
```

En la inicialización del nodo, path está vacío, todos los enteros están a 0.

Lista de nodos vivos

Para trabajar con los nodos vivos y establecer un orden de prioridad se ha hecho uso de un montículo de máximos priority queue. Esto permite obtener fácilmente el nodo más prometedor ya que siempre será el primero.

```
priority_queue<Node> vivos; //cola de prioridad.
```

```
vivos.push(nodo_inicial);//metemos el nodo inicial a la cola de prioridad
```

```
Node actual = vivos.top();      //cogemos el nodo prioritario, para ello cogemos el primer elemento de la cola de prioridad.
vivos.pop();//quitamos el nodo actual de los pendientes.
```

Mecanismo de poda

Poda de nodos no factibles

Un nodo se considera factible cuando nos encontramos en una casilla que no sea la inicial, ni la final y que, en el laberinto sea accesible. Es decir, hay un 1. Si no se cumplen estas condiciones, se descarta directamente el nodo.

```
//comprobacion si es factible
if(fila_siguiente >= 0 && fila_siguiente < datos.filas ){
    if(columna_siguiente >= 0 && columna_siguiente <= datos.columnas){
        if(datos.maze[fila_siguiente][columna_siguiente]!=0){
```

```
        }else{
        nunfeasible++;
        }
    }else{
        nunfeasible++;
    }
}else{
    nunfeasible++;
}
```

Poda de nodos no prometedores

Una vez determinado si es factible, se comprueba si el nodo es prometedor. Esto se hace comprobando que la longitud del camino desde el inicio hasta el hijo del nodo factible sea menor que la longitud del camino desde el inicio hasta el nodo factible. Si lo es, se analiza las cotas optimistas del nodo hijo y la mejor longitud obtenida hasta ese momento. Si la cota optimista del hijo es peor, no se añade a la cola de prioridad quedándose descartado.

```
//comprobacion si es prometedor
if(hijo_nodo_actual.path.size() + 1 < static_cast<unsigned int>(solutions[fila_siguiente][columna_siguiente])){  //
    solutions[fila_siguiente][columna_siguiente] = hijo_nodo_actual.path.size() + 1;

    if(datos.filas - hijo_nodo_actual.fila > datos.columnas - hijo_nodo_actual.columna){
        hijo_nodo_actual.optimista=hijo_nodo_actual.k + datos.filas - hijo_nodo_actual.fila;
    }
    else{
        hijo_nodo_actual.optimista=hijo_nodo_actual.k + datos.columnas - hijo_nodo_actual.columna;
    }

    if(hijo_nodo_actual.optimista < mejor_hasta_ahora){//si el hijo tiene buena cota optimista, metemos ese nodo a
        n_best_solution_updated_from_pessimistic_bound++;
        vivos.push(hijo_nodo_actual);
        nexplored++;
    }else{
        n_promising_but_discarded++;
    }

}else{
    n_not_promissing++;
}
```

Cotas pesimistas y optimistas

Cota pesimista inicial (inicialización)

La cota pesimista inicial viene proporcionada por la solución iterativa.

```
int mejor_hasta_ahora = mazeIterative(datos,mejorcamino);
```

```
int mazeIterative(const Datos &datos, vector<int> &mejorcamino){
    vector<vector<int>> almacen(datos.filas, vector<int>(datos.columnas, 0));
    if(datos.maze[0][0] == 0){
        almacen[0][0] = numeric_limits<int>::max();
    }
    else{
        almacen[0][0] = 1;
    }
    int i=1;
    while(i<datos.filas){
        if(datos.maze[i][0] == 0 || almacen[i-1][0] == numeric_limits<int>::max()){
            almacen[i][0] = numeric_limits<int>::max();
        }
        else{
            almacen[i][0] = almacen[i-1][0] + 1;
        }
        int j=1;
        while(j<datos.columnas){

            if(datos.maze[i][j] == 0){
                almacen[i][j] = numeric_limits<int>::max();
            }
            else{
                if(almacen[i-1][j-1] != numeric_limits<int>::max() || almacen[i-1][j] != numeric_limits<int>::max() || almacen[i][j-1] != numeric_limits<int>::max()){

                    int valor1 = almacen[i-1][j-1];
                    int valor2 = almacen[i-1][j];
                    int valor3 = almacen[i][j-1];
                    almacen[i][j] = (valor1 < valor2) ? ((valor1 < valor3) ? valor1 : valor3) : ((valor2 < valor3) ? valor2 : valor3) + 1;
                }
                else{

                    almacen[i][j] = numeric_limits<int>::max();
                }
            }
            if(datos.maze[0][j] == 0 || almacen[0][j-1] == numeric_limits<int>::max()){
                almacen[0][j] = numeric_limits<int>::max();
            }
            else{
                almacen[0][j] = almacen[0][j-1] + 1;
            }
            j++;
        }
        i++;
    }
    mejorCamino(datos,mejorcamino,almacen);
    return almacen[datos.filas - 1][datos.columnas - 1];
}
```

Cota pesimista del resto de nodos

Para el resto de los nodos se comprueba si la longitud del camino entre el inicio y ese nodo sea menor que aquella valor almacenada en la matriz solutions que inicialmente, se rellena con infinitos. Cuando es menor, se actualiza la matriz solutions.

```
vector<vector<int>> solutions(datos.filas, vector<int>(datos.columnas, numeric_limits<int>::max()))
```

```
if(hijo_nodo_actual.path.size() + 1 < static_cast<unsigned int>(solutions[fila_siguiente][columna_siguiente])){
    solutions[fila_siguiente][columna_siguiente] = hijo_nodo_actual.path.size() + 1;
```

Cota optimista

La cota optimista empleada es la distancia de Chebyshev

```
if(datos.filas - hijo_nodo_actual.fila > datos.columnas - hijo_nodo_actual.columna){
    hijo_nodo_actual.optimista=hijo_nodo_actual.k + datos.filas - hijo_nodo_actual.fila;
}
else{
    hijo_nodo_actual.optimista=hijo_nodo_actual.k + datos.columnas - hijo_nodo_actual.columna;
}

if(hijo_nodo_actual.optimista < mejor_hasta_ahora){//si el hijo tiene buena cota optimista, metemos
    n_best_solution_updated_from_pessimistic_bound++;
    vivos.push(hijo_nodo_actual);
    nexplored++;
```

Otros medios empleados para acelerar la búsqueda

Nada que comentar.

Estudio comparativo de distintas estrategias de búsqueda

Ahora se va a realizar una comparación entre ramificación y poda, pero cambiando lo que se usa como cota optimista. Estudiaremos la distancia Chebyshev y la distancia diagonal mínima

Distancia Chebyshev	Diagonal mínima	Fichero
62 9 1 0 6 0 2 0 0 1.233	62 9 1 0 6 0 2 0 0 1.21	16-bb
81 9 1 0 6 0 2 0 0 2.747	81 9 1 0 6 0 2 0 0 2.516	17-bb
106 9 1 0 5 0 3 0 0 2.853	106 9 1 0 5 0 3 0 0 3.287	18-bb
205 9 1 0 5 0 3 0 0 15.365	205 9 1 0 5 0 3 0 0 11.435	19-bb
3518 4070625 508829 1 1995254 1562635 3907 1 508829 530.13	3518 33755857 4219483 1 16593646 12930647 12081 1 4219483 7830.69	20-bb
5866 11228689 1403587 1 5508019 4310662 6421 1 1403587 2076.83	5866 93000129 11625017 1 45734839 35618635 21638 1 1162501 37243.3	21-bb
19 9 1 0 5 0 3 0 0 0.008	ERROR	22-bb
470 9 1 0 5 0 3 0 0 1.197	470 9 1 0 5 0 3 0 0 1.189	23-bb

Podemos averiguar con esta comparación que no se presentan muchas diferencias en la mayoría de casos, pero hay 3 casos que destacan.

En el archivo 20-bb la Distancia de Chebyshev expande muchos menos nodos que en la diagonal mínima. Esto afecta mucho al tiempo con una diferencia de 6500 ms

En el laberinto 21-bb ocurre lo mismo que en la 20-bb. Muchos menos nodos y una diferencia importante de tiempo. En este caso, 35000ms.

Por último, hay que mencionar lo ocurrido en el archivo 22-bb que es un error del sistema a la hora de usar la diagonal mínima.

Tiempos de ejecución

Fichero 16_bb.maze: 1.233 ms

Fichero 17_bb.maze: 2.747 ms

Fichero 18_bb.maze: 2.853 ms

Fichero 19_bb.maze: 15.365 ms

Fichero 20_bb.maze: 530.13 ms

Fichero 21_bb.maze: 2076.83 ms

Fichero 22_bb.maze: 0.008 ms

Fichero 23_bb.maze: 1.197 ms