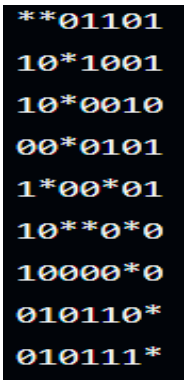
Análisis y diseño de algoritmos.

Práctica Final

Universitat d'Alacant
Universidad de Alicante

[El PROBLEMA DEL LABERINTO]



Nathan Rodríguez Moyses NRM69@ALU.UA.ES
GRUPO ARA

Índice:

1. Estructuras de datos

En este apartado se listan todas las versiones de nodos usadas.

1.1 Nodo

Para hacer este algoritmo (branch and bound sobre el problema del laberinto con 8 movimientos) he ido desde un algoritmo base y he ido añadiendo cotas y mejoras al algoritmo. La estructura para identificar al nodo no es un caso aparte en esta cuestión.

El nodo inicial tiene 3 enteros, la coordenada i, la j y el valor con el que se ha llegado. Adicionalmente, para evitar ciclos se guarda el menor valor con el que se ha llegado a cada nodo en una matriz de visitados que se inicializa a la matriz del algoritmo iterativo de programación dinámica sin mejora de memoria.

Esa es la primera versión, la segunda (y final) he añadido 1 entero más para poder explorar primero los nodos más prometedores, guardando la cota optimista de estos.

```
// 1
// using Node = tuple<int, int, int>;

//                  i    j  current opt
using Node = tuple<int, int, int, int>;
```

Como se puede apreciar uso tuple como contenedor para todas las variables del nodo.

1.2 Lista de nodos vivos

He probado varias formas de organizar los nodos, la primera versión era una cola de prioridad sin tener en cuenta la forma en la que se organizaban, por default lo obtenía en función del mayor coordenada i, lo que no tiene ningún propósito a la hora de mejorar la eficiencia.

Después, añadí la versión del nodo con la cota optimista y organicé la cola de prioridad con los menores de esta. Esta es la forma más eficiente de hacerlo que he encontrado, probé a usar una FIFO pero no resultó ser más eficiente que la ya mencionada, de todas formas los tiempos no son para nada malos con esta estructura.

Para programarlo he utilizado la std de c++ (priority_queue, queue).

```
// 1
// priority_queue<Node> pq;
```

Esta cola de prioridad es de máximos y la variable de la cual depende es la coordenada i, esto hace ineficiente el código por lo que se mejora con la siguiente versión.

```
struct is_worse{
  bool operator()(const Node&a, const Node&b){
    return get<3>(a) > get<3>(b);
  }
};
```

```
// 2
priority_queue<Node, vector<Node>, is_worse> pq;
```

Ahora, los nodos se extraen de menor a mayor en función de su cota pesimista (haciendo uso de la segunda, y última, versión presentada anteriormente). Esta es la versión definitiva que presenta mi programa.

Experimentando con otras formas de explorar los nodos, he probado la cola FIFO, que sorprendentemente tiene bastante eficiencia pero no llega a la versión presentada anteriormente.

```
// 3
// queue<Node> pq;
```

```
// 3
// auto [i, j, current, opt] = pq.front();
```

Nótese que en vez de usar el método top() debo usar front() para extraer los nodos, esta es la única diferencia que hay en el código usando una u otra (más allá de la velocidad de cómputo).

2. Mecanismos de poda

En este apartado se explican las formas que he llegado a usar para podar los nodos sin cotas.

2.1. Poda de nodos no factibles

Los nodos no factibles son aquellos que en su casilla tienen un 0 o se salen de la matriz. En la matriz de visitados se marca como infinito y no se llegan a incluir en el listado de nodos vivos.

```
for (int a = 1; a < 9; a++) {
  visited_nodes++;

  n = prioritize(a);
  a_i = i;
  a_j = j;

  int sol_aux = current;
  directions(n, a_i, a_j); // get the direction of the n

  // no feasible
  if (a_i < 0 || a_j < 0 ||
    a_j > END_J || a_i > END_I ||
    matrix[a_i][a_j] == 0) {
    no_feasible_discarded_nodes++;
    continue;
  }
```

El último if comprueba estas condiciones.

2.2. Poda de nodos no prometedores

Si el nodo tiene una cota optimista de mayor valor que la mejor solución encontrada hasta ahora no se expande el nodo. Esto incluye si hay ciclos, ya que usando la matriz de visitados se descartan los nodos que van hacia atrás.

```
// no cycles
if (arrived[a_i][a_j] <= sol_aux) {
  no_promissing_discarded_nodes++;
  continue;
}
```

```
if (sol <= sol_aux) {
  no_promissing_discarded_nodes++;
  continue;
}
```

La primera condición comprueba de que no hayan ciclos y ahorra bastante cómputo ya que está inicializada a la matriz resultante del algoritmo realizado en la práctica 6 la cual usa 3 movimientos.

3. Cotas pesimistas y optimistas

No he usado muchas cotas la verdad, este es el resumen de todas las que he probado.

3.1 Cota pesimista inicial (inicialización).

Uso el valor que obtiene el algoritmo iterativo de programación dinámica, esto inicializa la matriz de visitados y me da un valor inicial factible (mayoría de casos no, ya que no resuelve el problema) a la que mejorar. Inicializo la variable de mejor solución a este valor:

```
// 1
int sol = iterative_sol;
```

3.2 Cota pesimista del resto de nodos.

La idea de usar una cota pesimista en este problema es poder obtenerla en tiempo constante para todos los nodos, si no, no merece la pena.

No uso cota pesimista, es carga mucho el algoritmo intentar usar una cota pesimista, he probado el algoritmo voraz, pero no vale la pena perder para maximizar la eficiencia del algoritmo.

Otros algoritmos que he probado es usar un iterativo que calcule el valor mínimo que queda con 3 movimientos hasta el final, esto es válido ya que es un valor que actualiza (en algunos casos) la mejor solución encontrada pero que tampoco merece la pena realizar por el tiempo que se obtiene.

```
int vor = maze_greedy(matrix, i, j);
if ( vor!= numeric_limits<int>::max() && vor+current < sol) {
  sol = vor+current;
}
```

Esta técnica es inviable ya que el cálculo que se hace es pequeño pero en unas cantidades inviables, el último mapa propuesto son cantidades de millones de nodos, es demasiada carga de cómputo.

Como segunda opción, la que he mencionado antes; calcular con 3 movimientos (al igual que el iterativo) pero desde el final hasta la posición (0,0). Esto debería poder accederse en tiempo constante para todos los nodos y dar un valor seguro desde el nodo hasta el final del laberinto. Esto no ha funcionado como me esperaba, imaginaba que iba a podar bastante pero solo es rentable en las entradas más pequeñas. ¿A qué se debe esto? no lo se, lastimosamente debuggear este problema es bastante complejo, poda nodos en cantidad relativamente baja (funciona mejor esta versión en los problemas pequeños (todos menos el último, ver punto 5), en el último solo unos miles, esto significa que es positivo pero sale más a cuenta computar esos nodos que hacer esta preparación previa de los datos.

```
for (int i = rows - 1; i >= 0; i--) {
  for (int j = cols - 1; j >= 0; j--) {
    if (i == rows - 1 && j == cols - 1) {
      memo[i][j] = 1;
      continue;
    }

    if (matrix[i][j] == 0) {
      memo[i][j] = INF;
      continue;
    }


    int minNext = INF;

    if (i + 1 < rows) minNext = min(minNext, memo[i + 1][j]);        // Abajo
    if (j + 1 < cols ) minNext = min(minNext, memo[i][j + 1]);        // Derecha
    if (i + 1 < rows && j + 1 < cols) minNext = min(minNext, memo[i + 1][j + 1]); // Diagonal

    if (minNext != INF)
      memo[i][j] = 1 + minNext;
    else
      memo[i][j] = INF;
  }
}
```

3.3 Cota optimista.

La cota optimista que he usado es la distancia de Chebyshev, se calcula obteniendo la máxima de las diferencias desde las coordenadas hasta los límites de la matrices.

```
int optimistic_limit(int pos_i, int pos_j){
  int a_i = END_I - pos_i;
  int a_j = END_J - pos_j;

  return max(a_i, a_j);
}
```

4. Otros medios empleados para acelerar la búsqueda

He usado la matriz de visitados para evitar ciclos y poder podar en caso de llegar a un nodo el cual ya he llegado con menor o igual número de elementos. También he realizado una priorización de las direcciones a las que computo primero.

```
int prioritize(int i){
  switch (i) {
    case 1:
      return 4;
    case 2:
      return 5;
    case 4:
      return 1;
    case 5:
      return 2;
    default:
      return i;
  }

}
```

```
switch (n) {
  case 1:
    pos_i--;
    return;
  case 2:
    pos_i--;
    pos_j++;
    return;
  case 3:
    pos_j++;
    return;
  case 4:
    pos_i++;
    pos_j++;
    return;
  case 5:
    pos_i++;
    return;
  case 6:
    pos_i++;
    pos_j--;
    return;
  case 7:
    pos_j--;
    return;
  case 8:
    pos_i--;
    pos_j--;
    return;
  default:
    return;
}
```

La función de la izquierda prioriza el movimiento y el de la derecha calcula el valor por referencia de la dirección en la matriz (este último no mejora nada, simplemente calcula las coordenadas que se requieren en cada caso)

5. Estudio comparativo de distintas estrategias de búsqueda

Como selección de nodos he probado la cola FIFO, el montículo de prioridad de mínimos respecto la cota optimista y el montículo de prioridad de máximos con la cota optimista. La que mejor me ha funcionado es el montículo de mínimos respecto a la cota pesimista. La implementación se ha realizado sobreescribiendo el operador operator() tal como se muestra en las transparencias dadas en clase.

Respecto a las comparativas se van a realizar numerosas de ellas, respecto los tiempos, cada versión se va a ejecutar un total de 30 veces (y se va ha hacer la media aritmética) para tener una medida que merezca la pena ser estudiada, sin esta repetición no tendría sentido los tiempos dados. Esto se ha hecho con un shell script que ejecuta y guarda en un fichero los tiempos.

El primero que se va a ejecutar es el que no tiene la matriz de visitados inicializada, no tiene cota pesimista inicial, no tiene cota pesimista ni optimista para cada nodo, la cola de prioridad está respecto la variable “i”. Esta versión es de las que va a partir cualquier solución, es decir, es la versión que resuelve el problema más lento que se va a presentar.

El siguiente, se va inicializar la matriz de visitados y la mejor solución encontrada es la de programación dinámica iterativa.

En la tercera se van a añadir las cotas, usando la distancia de chebyshev para descartar los nodos que tengan peor cota optimista que la solución ya encontrada.

En la cuarta se cambia la estrategia de búsqueda y se añade la cola de prioridad de mínimos y el cambio de estructura de nodo, dejando la cola de prioridad de mínimos respecto a la cota optimista de cada nodo.

En la quinta se cambia la estrategia de búsqueda por un FIFO.

En la sexta y séptima se prueban diferentes cotas pesimistas, en la sexta el algoritmo voraz y en la séptima se prueba la iterativa al revés que mencioné previamente.

6. Tiempos de ejecución.

Tabla de tiempos:

versiones	7	8	9	10	11	12	13	100	200	300	500	700	900	k01	k02	k03	k05	k10
v1	0,02	0,04	0,04	0,05	0,27	0,01	23.297,10	78,48	?	?	?	?	?	?	?	?	?	?
v2	0,00	0,03	0,03	0,03	0,19	0,01	22.252,82	77,08	?	?	?	?	?	?	?	?	?	?
v3	0,01	0,01	0,02	0,03	0,05	0,01	5,37	3,90	5,87	888,14	327,38	2.999,31	4.159,63	?	?	?	?	?
v4	0,00	0,00	0,00	0,01	0,02	0,00	1,20	0,30	0,64	4,73	2,83	15,98	27,88	19,41	48,61	137,77	405,34	1.624,61
v5	0,01	0,01	0,01	0,02	0,05	0,01	13,98	0,72	2,57	6,15	2,95	35,81	59,69	69,23	48,98	649,68	2.004,22	3.742,80
v6	0,01	0,01	0,02	0,05	0,01	0,01	5,40	0,50	1,54	7,13	3,96	26,71	45,83	40,41	65,85	306,15	877,60	2.683,24
v7	0,01	0,00	0,01	0,02	0,03	0,00	5,01	0,27	1,07	6,88	3,76	26,32	46,92	24,80	64,18	304,21	901,22	2.866,72

El eje x representa la talla de los problemas que se resuelven y el eje y la versión usada en ese momento. Todas los problemas se han ejecutado 30 veces, se han sumado sus valores y dividido entre 30 para que tenga veracidad la muestra tomada.

La versión presentada es la versión 4, sin cotas pesimistas para cada nodo. Esta es la más eficiente.

7. Conclusión

El fichero escrito tiene todo el código probado, para elegir una versión es cuestión de comentar cada parte por separado. Para obtener el vector se realiza un parse de la matriz resultante de visitados. Esto no se tiene en cuenta en los tiempos mostrados en la tabla. Este parse es de la misma naturaleza que el de programación dinámica de memorización solo que con 8 movimientos.

Otras técnicas como multiplicar la cota optimista por un valor para maximizar la cantidad de nodos podados se ha probado pero no ha sido eficiente, esto, hipotéticamente, se debe a que la ALU del computador debe hacer más conversiones debido al truncamiento de los valores.

Lastimosamente no se mejoran los tiempos mostrados en las soluciones. Esto en gran medida (y por la cantidad de nodos que se podan y que no) se debe a la ausencia de cota pesimista en mi programa ya que no se llega a actualizar nunca (de forma prematura) la mejor solución encontrada. Es una lástima ya que he probado varias pero ninguna ha sido óptima.