Algoritmos y Estructuras de Datos
Tema 2: Diseño de Algoritmos

Contenidos

Algoritmos de vuelta atrás

- Algoritmos de ***vuelta atrás*** (backtraking) es un tipo general de algoritmos orientado a la resolución de problemas, que se suele aplicar en: *optimización*, *resolución de juegos*, *búsquedas de caminos*
- Se realizan **búsquedas exhaustivas** de entre todas las posibles soluciones
- Se **exploran todas las hipótesis**, guardando estado actual. Si no se da con la solución, se recupera el estado, y se explora otra
- La **recursividad** se suele emplear para **guardar estado** y realizar exploraciones recursivas

Algoritmos de vuelta atrás

- La **solución** son un conjunto de valores x_1, x_2, ..., x_n (por ejemplo, movimientos de las fichas de un juego) que satisfacen las restricciones del problema
  - Pueden ser una **solución cualquiera**
  - Pueden ser una **solución** que **optimizan** una función objetivo
- En un momento de la ejecución, el algoritmo puede tener una solución parcial de k valores (x_1, ..., x_k), y nos falta por encontrar n-k valores:
  - Nos encontramos en un **nivel k**
  - Se pasa a buscar un nuevo valor k+1 que satisfaga las restricciones, añadido a los anteriores k valores
    - Si no encontramos un k+1, deshacemos el x_k y buscamos una nueva solución x_k

Ideas básicas

- Debemos ser capaces de **enumerar todas las alternativas/ movimientos** posibles
  - **Se intentan todas** esas alternativas en un cierto orden, seleccionando una de ellas antes de seguir
  - Si la selección hecha no nos lleva a una solución posible, **hay que deshacerla**
    - Esto es **vuelta atrás**, las elecciones hechas se deben poder **deshacer**
- El proceso es intrínsecamente recursivo. Debemos saber **cuando termina** (*Ejecución Base*)
  - Saber cuando el problema está resuelto
  - Saber cuando no podemos seguir por ningún camino
- **En resumen**: enumerar los posibles siguientes pasos; intentar cada uno de ellos; deshacerlo en un punto muerto
  - **Fuerza bruta:** lo intentamos todo

Ejemplos: sudoku de 2x2

- Rellenar las 4 cajas de 2x2 con números de 1 al 4 no repetidos ni en las cajas, ni en las filas o columnas de 4x4

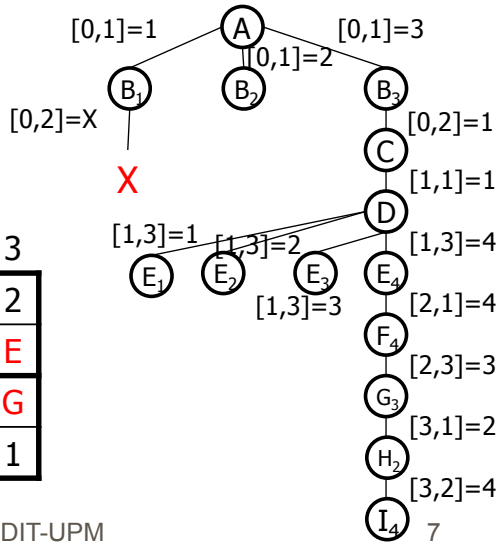
4			2
2		3	
1		2	
3			1

nivel A *nivel B* *nivel C* *nivel D* *nivel E* *nivel F* *nivel G*

Estructura general de los algoritmos

- Las soluciones que se van construyendo se conceptualizan como árboles ***implícitos*** de posibles soluciones:
  - Cada ***rama*** es un valor para un *paso hacía la solución*
  - Las hojas representan *soluciones posibles* o *puntos muertos*

A	0	1	2	3
0	4	B	C	2
1	2	D	3	E
2	1	F	2	G
3	3	H	I	1



Antes de implementar …(1)

- Antes de implementar debemos tener claro:
  1. Datos con los que **representamos la solución**
     - *Sudoku:* array dos dimensiones con los contenidos de posiciones
  2. Como **representamos** cada valor x_i (un **ensayo**)
     - *Sudoku:* fila, columna, valor
  3. **Restricciones** que limitan las **soluciones**, y como validar soluciones
     - *Sudoku*: No coincidencias caja, no coincidencias fila, no coincidencias columna
  4. Como se **genera** un **nuevo nivel**
     - *Sudoku*: Saltar a la siguiente casilla y fijar un primer valor

Antes de implementar …(2)

- Antes de implementar debemos tener claro:
  5. Como se **genera** un nuevo **hermano**
     - *Sudoku*: Nuevo intento de valor para la casilla para la que buscamos valor
  6. **Como** se **retrocede** en el árbol
     - Sudoku: Marcamos casilla como vacía y volvemos diciendo que es punto muerto
  7. Descartar ramas para **optimizar**
     - *Sudoku*: Vuelta atrás en los puntos muertos sin seguir explorando
     - Un sudoku válido tiene una única solución

Algoritmo general recursivo

```
recursivo_vuelta_atras(ensayo)
  if esUnaSoluciónCompleta(ensayo)
    return ensayoOK(ensayo)
  else
    if cumpleRestricciones(ensayo)
      for nuevoNivel ∈ siguientesNieveles(ensayo)
        if recursivo_vuelta_atras(nuevoNivel) == ensayoOK
          return ensayoOK
      for unEnsayo ∈ siguientesHermanosDelNivel(ensayo)
        deshacer ensayo y actualizar unEnsayo
        if recursivo_vuelta_atras(unEnsayo) == ensayoOK
          return ensayoOK(unEnsayo)
  deshacer ensayo
  return ensayoKO(ensayo)
}
```

Ejecución Base

Ejecución Recursiva

Cálculos Generales

Algoritmo sudoku recursivo

- Variables:
  - *int[][]* ***model***: array con los valores de las posiciones, inicializado con valores iniciales
  - *int* ***row, col***: fila y columna para la que buscamos valor
  - *int* ***num***: valor de la posición *row,col*
- Métodos:
  - ***checkRow****(int row, int num)*: comprueba que ningún elemento de de la fila *row* de *model* tiene asignado valor **num**
  - ***checkCol****(int col, int num)*: comprueba que ningún elemento de de la columna *col* de *model* tiene asignado valor **num**
  - ***checkBox****(int row, int col, int num)*: comprueba que ningún elemento de la caja en la que se cuenta la casilla *row, col* tiene asignado valor *num*
  - ***solve****(int row, int col):* rellena el sudoku que tenemos en *model* partiendo de la posición *row,col*

Algoritmo sudoku recursivo

```
protected int model[][] ;
protected boolean checkRow(int row, int num){
    for( int col = 0; col < COLUMNS; col++ )
        if( model[row][col] == num ) return false ;
    return true ;
}

protected boolean checkCol(int col, int num) {
    for( int row = 0; row < ROWS; row++ )
        if( model[row][col] == num ) return false ;
      return true ;
}

protected boolean checkBox(int row, int col, int num) {
    row = (row / BOX_ROWS) * BOX_ROWS ;
    col = (col / BOX_COLUMNS) * BOX_COLUMNS ;
    for( int r = 0; r < BOX_ROWS; r++ )
        for( int c = 0; c < BOX_COLUMNS; c++ )
            if( model[row+r][col+c] == num ) return false ;
    return true ;
}
```

Algoritmo sudoku recursivo

```
protected int model[][] ;
public boolean solve(int row, int col) {
      if( row > ROWS-1 ) // todas posiciones ocupadas y chequeadas
         return true;
      if( model[row][col] != 0 ) // posición ocupada-> siguiente
          return solve(row+((col+1)/COLUMNS), (col+1) % COLUMNS);
      else {
         for (int num = 1; num <= ROWS; num++) { // probamos hermanos
            if(checkRow(row,num) && checkCol(col,num) &&
               checkBox(row,col,num)){
               model[row][col] = num ;
               if (solve(row+((col+1)/COLUMNS), (col+1) % COLUMNS))
                 return true ;
            }
          }
          // Este es un punto muerto-> vuelta atrás
          model[row][col] = 0 ;
          return false;
       }
}
```


Algunas variantes

- **No** es seguro que **exista solución**
  - Ejemplo: Hay sudoku sin soluciones
  - La función debe devolver ambas alternativas
- Hay **múltiples soluciones** y queremos verlas **todas**
  - Ejemplo: Posibles caminos entre dos puntos de un mapa
  - Hay que guardarlas y seguir buscando
- Es un problema de optimización y buscamos la **solución que optimiza una función**
  - Ejemplo: Buscar camino mas corto de un mapa. Pueden ser varias soluciones, pero debemos optimizar una función
  - Hay que **guardar la óptima y seguir** buscando
  - La función de optimización nos puede servir para **abandonar soluciones parciales** que no podrán mejorar solución óptima

Heurísticas

- Una heurística es una regla que a *groso modo* nos permite **elegir/descartar** entre **alternativas**. Útiles en la mayoría de los casos, pero …
  - No siempre mejoran la elección
  - No siempre garantizan que sean la mejor forma de elegir
  - Su ejecución sobrecarga los algoritmos
- Heurísticas para sudoku para intentar menos hermanos (http://en.wikipedia.org/wiki/Algorithmics_of_sudoku)
  - Empezamos rellenando la casilla con menor número de posiciones vacías en su fila, columna o caja
  - Empezamos rellenando la casilla que tiene menor número total de casillas vacías en fila, columna y caja

Vuelta atrás

- Es un **recorrido exhaustivo** y sistemático de un árbol de soluciones
- Para aplicarlo debemos saber:
  - Como **identificar** cuando una **solución** está encontrada
  - **Avanzar** al siguiente paso
  - Recorrer **todas** las **alternativas** de un nivel
  - **Deshacer** pasos dados
- El tiempo de ejecución es proporcional al número de nodos del árbol x el tiempo de ejecución de cada uno de ellos (si es igual para todos los nodos)
- **Ejercicio:** ¿cuántos nodos puede llegar a tener un sudoku 9x9 con X valores iniciales?
- **En general**, los algoritmos de vuelta atrás tienen complejidad de orden **exponencial** o **factorial**
  - No usar si existen otras alternativas más rápidas
  - No son algoritmos escalables