EI1022, Algoritmia

Problemas 4 y 5: Búsqueda con retroceso

Introducción

Vamos a practicar la técnica de búsqueda con retroceso (*backtracking*) con dos problemas combinatorios:

- Resolución de sudokus
- Subconjunto de suma dada

Sudokus

- Un sudoku es un tablero de 9×9 dividido en bloques de 3×3 casillas, algunas de las cuales contienen números entre 1 y 9
- El objetivo es poner un número entre 1 y 9 en cada casilla vacía de modo que cada fila, columna y cada bloque del tablero contenga los 9 números. Un ejemplo de sudoku:

			3	1	6		5	9
		6				8		7
						2		
	5			3			9	
7	9		6		2		1	8
	1			8			4	
		8						
3		9				6		
5	6		8	4	7			

Representación de un sudoku

- Definimos el tipo `Sudoku` como:

```
type Sudoku = list[list[int]]
```

- Nuestro sudoku de ejemplo se representa así:

```
sudoku = [[0, 0, 0, 3, 1, 6, 0, 5, 9],
          [0, 0, 6, 0, 0, 0, 8, 0, 7],
          [0, 0, 0, 0, 0, 0, 2, 0, 0],
          [0, 5, 0, 0, 3, 0, 0, 9, 0],
          [7, 9, 0, 6, 0, 2, 0, 1, 8],
          [0, 1, 0, 0, 8, 0, 0, 4, 0],
          [0, 0, 8, 0, 0, 0, 0, 0, 0],
          [3, 0, 9, 0, 0, 0, 6, 0, 0],
          [5, 6, 0, 8, 4, 7, 0, 0, 0]]
```

- El `0` representa una celda vacía.

- Utilizaremos el mismo tipo `Sudoku` para las soluciones, con la diferencia de que no tendrán ningún cero.

El programa `sudoku.py`

- Debéis implementar el programa `sudoku.py` que resolverá el sudoku que reciba por la entrada estándar
- El programa seguirá el esquema de la asignatura y usará tanto el módulo `bt_scheme` de la biblioteca algoritmia como el fichero `sudoku_lib.py` disponible en el aula virtual

Lectura de datos

- La función `read_data` leerá de `f` un sudoku representado como una serie de líneas, una por cada fila
- Por lo tanto, el tipo `Data` sera:

  ```
  type Data = Sudoku
  ```

- Las filas serán cadenas en las que las casillas con dígitos se representarán con el carácter correspondiente y las casillas vacías con un punto

Lectura de datos (2)

- Por ejemplo, el sudoku de la trasparencia 3 se representa así:

```
...316.59
..6...8.7
......2..
.5..3..9.
79.6.2.18
.1..8..4.
..8......
3.9...6..
56.847...
```

Lectura de datos (3)

- Para facilitaros la implementación de `read_data`, podéis usar la función `from_strings` de `sudoku_lib.py`
- Su firma es

```
def from_strings(strings: Iterable[str]) -> Sudoku:
```

- Se le pasa una lista con las cadenas leídas del fichero y devuelve el sudoku correspondiente

La función `process`

- La función `process` recibirá un sudoku y devolverá todas las posibles soluciones mediante un iterador
- Por lo tanto, el tipo `Result` será:

```
type Result = Iterator[Sudoku]
```

- Para encontrar las soluciones utilizará la función `bt_solutions` de `bt_scheme`

Función bt_solutions

- La función bt_solutions es

```
def bt_solutions[D, E](ds: DecisionSequence[D, E]) -> Iterator[DecisionSequence[D, E]]:
    if ds.is_solution():
        yield ds
    for new_ds in ds.successors():
        yield from bt_solutions(new_ds)
```

- Para utilizarla, tendréis que declarar una clase que herede de DecisionSequence:

```
class SudokuDS(DecisionSequence[Decision, Extra]):
```

La clase SudokuDS

- Una decisión es colocar un número en una celda representada por su fila y columna:

  ```
  type Decision = tuple[Position, int]
  ```

  donde `Position` es un tipo importado de `sudoku_lib`

- Con la secuencia de decisiones es muy difícil averiguar de forma eficiente si el sudoku es una solución o generar sus sucesores, así que utilizaremos la clase `Extra` para almacenar el sudoku

La clase `SudokuDS` (2)

- Los métodos que hay que implementar serán:
  - `is_solution` que devolverá `True` si el sudoku no tiene ninguna casilla vacía
  - `successors` que generará los posibles sucesores del sudoku

Sobre el método `successors`

Podemos considerar dos implementaciones:

- Probar todos los valores posibles en todas las casillas vacías. Problemas:
  - Muchísimas secuencias de decisiones generarán los mismos sudokus
  - El tamaño del árbol de búsqueda será innecesariamente grande
- Probar todos los valores posibles pero sólo en la primera casilla vacía:
  - Dos secuencias de decisiones diferentes siempre generan sudokus diferentes
  - Al no contener repetidos, nuestro árbol de búsqueda será mucho menor

Funciones auxiliares

- Para encontrar la primera casilla vacía podéis usar la función `first_empty`, que se implementa usando la función `empty_cells`, ambas en `sudoku_lib.py`
- Para saber qué números se pueden poner en una casilla, podéis utilizar la función `allowed`, también de `sudoku_lib.py`

Funciones `empty_cells` y `first_empty`

Devuelven las casillas vacías y la primera de ellas, respectivamente:

```
def empty_cells(s: Sudoku) -> Iterator[Position]:
    for row in range(9):
        for col in range(9):
            if s[row][col] == 0:
                yield row, col

def first_empty(s: Sudoku) -> Position | None:
    return next(empty_cells(s), None)
```

Función auxiliar `allowed`

Devuelve el conjunto de los números que se pueden poner en una casilla:

```
def allowed(s: Sudoku, pos: Position) -> set[int]:
    row, col = pos
    fc, cc = row // 3 * 3, col // 3 * 3
    used = ({s[row][c] for c in range(9)}
            | {s[f][col] for f in range(9)}
            | {s[fc + f][cc + c]
               for f in range(3)
               for c in range(3)})
    return set(range(1, 10)) - used
```

Escritura de resultados

- La función `show_result` recibirá el iterador producido por `process` y mostrará cada uno de los sudokus
- Para que os sea más fácil escribirlo, en `sudoku_lib.py` tenéis la función

  ```
  def pretty_print(s: Sudoku):
  ```

  que muestra por pantalla el sudoku que recibe como parámetro

Prueba del programa

- Una vez lo tengáis implementado, ejecutad
  `python3 sudoku.py < sudokus/normal.sku`
- `normal.sku` esta en el aula virtual y corresponde al sudoku de la trasparencia 3
- La salida debe ser el sudoku

2	8	7	3	1	6	4	5	9
1	4	6	2	9	5	8	3	7
9	3	5	4	7	8	2	6	1
8	5	2	1	3	4	7	9	6
7	9	4	6	5	2	3	1	8
6	1	3	7	8	9	5	4	2
4	2	8	9	6	3	1	7	5
3	7	9	5	2	1	6	8	4
5	6	1	8	4	7	9	2	3

Eficiencia

- Nuestra implementación resuelve rápidamente `normal.sku`
- ¿Y si probamos con el sudoku más difícil del mundo (fichero `sudokus/dificil.sku`?

8								
		3	6					
	7			9		2		
	5				7			
				4	5	7		
			1				3	
		1					6	8
		8	5				1	
	9					4		

- Encuentra rápidamente la solución, pero tarda mucho en terminar de explorar el espacio de estados para ver que no hay ninguna más.

Eficiencia (2)

- Copia `process` en `process_fast` y renombra la versión antigua a `process_slow`
- Cambia el main para poder elegir entre las dos versiones:

```
if __name__ == "__main__":
    # process = process_slow
    process = process_fast
    data = read_data(sys.stdin)
    result = process(data)
    show_result(result)
```

Eficiencia (3)

- Implementa las siguientes modificaciónes en `process_fast`:
  - Añade un atributo adicional a la clase `Extra` para guardar el conjunto de posiciones vacías del Sudoku
  - Modifica `successors` para que elija el elemento del conjunto de posiciones vacías que tenga menos números posibles
  - Modifica el método `is_solution` para que se limite a comprobar que no quedan celdas vacías
- Mejora adicional: modifica el programa para que no se hagan copias del sudoku ni del conjunto de posiciones vacías (reutiliza la clase `Extra`)

El problema de la suma del subconjunto

- Tenemos una lista no vacía $e = (e_0, \ldots, e_{n-1})$ de números naturales distintos de 0
- Podemos elegir un subconjunto de índices de e y obtener la suma de los elementos correspondientes a esos índices
- Esto nos permite plantear dos problemas:
  1. ¿Podemos elegir los índices de modo que la suma sea un valor S dado?
  2. Si se puede, ¿cuál es el subconjunto de índices de mínimo tamaño?

El problema de la suma del subconjunto (2)

- Podemos representar el conjunto de índices asociando a cada uno de los e_i un valor x_i que sea 0 si e_i no está en el conjunto y 1 si lo está
- La suma de los elementos correspondientes a los índices del conjunto será: $\sum_{i=0}^{n-1} x_i e_i = S$
- Así, el conjunto de soluciones factibles será:

$$X = \left\{ (x_0, \ldots, x_{n-1}) \in \{0,1\}^n \,\middle|\, \sum_{i=0}^{n-1} x_i e_i = S \right\}$$

Ejercicio 1: Función objetivo

- Piensa cómo podemos expresar el número de elementos que tiene el conjunto de índices en función de los x_i
- Escribe la función objetivo para el problema 2 (buscar el subconjunto con el mínimo número de elementos)

El programa `subset_sum.py`

- Escribiremos el programa `subset_sum.py` para resolver el problema de la suma del subconjunto
- La entrada (que leerá `read_data`) tendrá una línea con el valor de S, seguida de n líneas una por cada e_i
- El tipo `Data` será:

```
type Data = tuple[int, list[int]]
```

La función `process`

- La función `process` devolverá `None` o una lista con los e_i tales que $x_i = 1$
- El tipo `Result` será:

```
type Result = list[int] | None
```

- Dentro de `process` deberás definir las clases `Extra` y `SubsetSumDS`
- En la clase `Extra` nos bastará con tener la suma de los elementos hasta el momento
- La clase `SubsetSumDS` heredará de `DecisionSequence` y deberá implementar los métodos `is_solution` y `successors`

La función `process` (2)

- Para recorrer el espacio de estados, debes usar la función `bt_solutions`
- Como estamos buscando la mejor solución, podemos utilizar la función `min_solution`, pero hay que tener en cuenta que debemos procesar el valor devuelto para ajustarlo al formato deseado:
  Por ejemplo, para `S = 14` y `e = [1, 2, 3, 4, 5]`, el valor devuelto por `min_solution` es `(4, (0, 1, 1, 1, 1))` pero `process` debe devolver `[2, 3, 4, 5]`

La función `show_results`

- Lo que muestre `show_result` dependerá del tipo de su parámetro:
  - Si recibe `None`, escribirá `"No hay solución"`
  - Si recibe una lista, la escribirá con un número por línea

Prueba del programa

- En el directorio `sets` tenéis varios ficheros de prueba:
  - `example.set` es el ejemplo que hemos comentado al hablar de `process`
  - El nombre del resto tiene:
    - El tamaño: `small`, `medium` o `large`
    - El valor de S
    - El tipo `.set` para el problema, `.sol` para la solución
- `subset_sum.py` resuelve rápido todos los problemas `small`, `medium_1209`, `medium_3531` y `large_742`
- En el resto, tarda mucho

Control de visitados

- Vamos a intentar resolver los problemas de tamaño medio y grande
- En un primer paso, nos conformaremos con resolver el problema 1 (¿hay un subconjunto de la suma dada?) sin garantizar encontrar la solución óptima
- Copia `process` en `process_state`, renombra el original a `process_simple` y cambia el main para poder elegir entre las dos

Control de visitados (2)

- Modifica `process_state` para que utilice `bt_vc_solutions` en lugar de `bt_solutions`
- Tendrás que sobreescribir el método `state` de `SubsetSumDS`
- Un `state` razonable contiene la suma parcial y la longitud de la secuencia de decisiones
- Prueba ahora a resolver todos los problemas

Garantizar la solución óptima

- Al resolver los problemas habrás visto que `process_state` no devuelve la solución óptima, p.ej. para `medium_3531.set` devuelve un subconjunto de tamaño 6 cuando la solución óptima tiene tamaño 5
- Copia `process_state` en `process_full_state`
- Cambia el main para poder elegir entre las tres versiones
- Modifica `process_full_state` para garantizar que encuentra la solución óptima