Prácticas de Algorítmica

Práctica 2: Backtracking

Curso 2022-2023

Introducción a *backtracking*

Introducción a *backtracking* vía subset sum

- La búsqueda con retroceso o *backtracking* es un proceso de búsqueda de una solución factible en un espacio que se puede ver como una secuencia de decisiones.
- Es decir, cualquier solución **factible** es tan buena como las demás: no se trata de buscar la que mejore cierta función objetivo, no es un problema de optimización sino de factibilidad.
- Veamos el problema de la suma de subconjuntos `https://en.wikipedia.org/wiki/Subset_sum_problem`.
  La entrada es un vector de números positivos `valores = [5, 7, 12, 30, 40, 15, 20, 9]` y un `objetivo = 49`. Se trata encontrar un subconjunto de valores que sume el objetivo.

Nota (sobre programación dinámica)

Es posible abordar el *subset sum* con programación dinámica, pero el coste es *pseudo-polinómico:* coste polinómico con el valor de la variable `objetivo` que no es la talla del problema.

Introducción a *backtracking* vía subset sum

- Este problema se parece a la mochila discreta pero se trata de llenar la mochila a tope y sin beneficios.
- Una solución se puede representar con un vector de 0s y 1s de longitud `len(valores)` donde 0 significa que nos saltamos el número y 1 que lo sumamos.
- El siguiente esquema simula un recorrido primero en profundidad en un árbol binario completo de profundidad `len(valores)` donde cada hoja representa una posible solución (en este caso factible o no):

```python
def recorridoRecursivo(valores, sol):
    if len(sol) == len(valores):
        # hemos llegado a una hoja
        print(sol)
    else:
        for opcion in [1,0]:
            recorridoRecursivo(valores, sol+[opcion])

valores = [5, 7, 12, 30] # ponemos un ejemplo más cortito
```

Subset sum por fuerza bruta

La salida de `recorridoRecursivo(valores, [])` es:

```
[1, 1, 1, 1]
[1, 1, 1, 0]
[1, 1, 0, 1]
[1, 1, 0, 0]
[1, 0, 1, 1]
[1, 0, 1, 0]
[1, 0, 0, 1]
[1, 0, 0, 0]
[0, 1, 1, 1]
[0, 1, 1, 0]
[0, 1, 0, 1]
[0, 1, 0, 0]
[0, 0, 1, 1]
[0, 0, 1, 0]
[0, 0, 0, 1]
[0, 0, 0, 0]
```

Subset sum por fuerza bruta

Podemos resolver la suma de subconjuntos mediante **fuerza bruta** (también llamado **generate and test**):

```
def sumaSubcjtsFuerzaBruta(valores, objetivo, sol):
    if len(sol) == len(valores):
        # hemos llegado a una hoja
        suma = sum(v for v,s in zip(valores,sol) if s==1)
        if suma == objetivo:
            print(list(v for v,s in zip(valores,sol) if s==1))
    else:
        for opcion in [1,0]:
            sumaSubcjtsFuerzaBruta(valores, objetivo, sol+[opcion])

sumaSubcjtsFuerzaBruta([5, 7, 12, 30, 40, 15, 20, 9], 49, [])
```

Que da el siguiente resultado:

```
[5, 15, 20, 9]
[7, 12, 30]
[40, 9]
```

Pero tiene un coste $\theta(2^N)$ siendo N el nº valores.

Uso de una función local o *closure*

Antes de mejorar este recorrido básico vamos a quitar el argumento `valores`, a dejar fijo el vector `sol` y a utilizar una clausura o *closure* (una función dentro de otra):

```
def sumaSubcjtsFuerzaBruta(valores, objetivo):
  N = len(valores)
  sol = [None]*N
  def backtracking(longSol):
    if longSol == N: # hemos llegado a una hoja
        suma = sum(v for v,s in zip(valores,sol) if s==1)
        if suma == objetivo:
            print(list(v for v,s in zip(valores,sol) if s==1))
    else:
        for opcion in [1,0]:
            solucion[longSol] = opcion
            backtracking(longSol+1)
  backtracking(0)

sumaSubcjtsFuerzaBruta([5, 7, 12, 30, 40, 15, 20, 9], 49)
```

Subset sum con poda por factibilidad

Ahora vamos a bajar únicamente por aquellas soluciones que sean prometedoras, es decir:

- Si añado un 0 debo asegurarme de que el peso que llevo hasta ahora más el peso de lo que puedo llegar a meter en la mochila (añadir siempre lo que queda por decidir) es capaz de alcanzar el valor deseado.
- Si añado un 1 debo comprobar que no me haya pasado de peso.

Poda por factibilidad

No explorar ciertas partes del árbol se denomina **poda por factibilidad** y es lo que convierte esto en backtracking que es MUCHO más eficiente que la *fuerza bruta* (o *generate and test*).

Subset sum con poda por factibilidad

Hemos desenrollado el bucle que ramifica porque cada caso es diferente:

```
def subsetSum(valores, objetivo):
  N = len(valores)
  sol = [None]*N
  def backtracking(longSol, pesoAcumulado):
    if longSol == N: # hemos llegado a una hoja
      if pesoAcumulado == objetivo: # me sirve?
        print([valores[i] for i in range(N) if sol[i]==1])
    else: # es nodo interno, vamos a RAMIFICAR
      w = valores[longSol] # peso a considerar en esta etapa
      if pesoAcumulado + w <= objetivo: # si no me paso
        sol[longSol] = 1
        backtracking(longSol+1, pesoAcumulado+w) # acumulo peso
      # podré alcanzar el objetivo?
      if pesoAcumulado + sum(valores[longSol+1:]) >= objetivo:
        sol[longSol] = 0
        backtracking(longSol+1, pesoAcumulado) # NO acumulo peso
  backtracking(0,0) # longitud 0 y peso acumulado 0
```

Subset sum con poda por factibilidad

Ejemplo de ejecución:

```
valores = [5, 7, 12, 30, 40, 15, 20, 9]
objetivo = 49
subsetSum(valores, objetivo)
```

El resultado es nuevamente:

```
[5, 15, 20, 9]
[7, 12, 30]
[40, 9]
```

Mejora del precálculo

Podemos eliminar `sum(valores[longSol+1:])` con un precálculo:

```python
def subsetSum2(valores, objetivo):
  N = len(valores)
  sol, pesoDerecha = [None]*N, [0]*N
  for i in range(N-2,-1,-1): # desde penúltimo hasta primero
    pesoDerecha[i] = pesoDerecha[i+1] + valores[i+1]
  def backtracking(longSol, pesoAcum):
    if longSol == N: # hemos llegado a una hoja
      if pesoAcum == objetivo: # me sirve la solución?
        print([valores[i] for i in range(N) if sol[i]==1])
    else: # es nodo interno, vamos a RAMIFICAR
      w = valores[longSol] # peso a considerar
      if pesoAcum + w <= objetivo: # si no me paso
        sol[longSol] = 1
        backtracking(longSol+1, pesoAcum+w) # acumulo
      if pesoAcum + pesoDerecha[longSol] >= objetivo:
        sol[longSol] = 0
        backtracking(longSol+1, pesoAcum) # no acumulo
  backtracking(0,0) # longitud 0 y peso acum 0
```

Cuestión

¿Hace falta ver si es factible una solución siempre prometedora?
Prueba a eliminar esta línea:

```
if pesoAcumulado == objetivo: # me sirve esta solución?
```

¿Sigue funcionando bien o da otras soluciones?

Sigue funcionando porque si llega a un estado terminal (condición `longSol == N`) significa que tras haber considerado todos los objetos el peso de la parte acumulada es $\leq W$ y al mismo tiempo el peso del a parte acumulada más la parte restante (que, al estar vacía, es 0) es $\geq W$. Por tanto, al ser al mismo tiempo mayor o igual y menor o igual a W, necesariamente es igual a W.

Regla general

Si no podamos por factibilidad decimos que el estado es *prometedor*. En muchos casos (pero no en todos) se puede demostrar que si es *prometedor* y *terminal* implica que también es *factible*, en ese caso eliminamos el `if factible`.

Parar tras encontrar una solución

NO basta con usar `return` donde teníamos el `print`, hay que **propagar esa parada en el resto de llamadas recursivas**

```
...
def backtracking(longSol, pesoAcum):
  if longSol == N: # hemos llegado a una hoja
    return (sol,[valores[i] for i in range(N) if sol[i]==1])
  else: # es nodo interno, vamos a RAMIFICAR
    w = valores[longSol] # peso a considerar en esta etapa
    if pesoAcum + w <= objetivo: # si no me paso
      sol[longSol] = 1
      resul = backtracking(longSol+1, pesoAcum+w)
      if resul is not None:
        return resul
    if pesoAcum + pesoDerecha[longSol] >= objetivo:
      sol[longSol] = 0
      resul = backtracking(longSol+1, pesoAcum)
      if resul is not None:
        return resul
return backtracking(0,0) # longitud 0 y peso acum 0
```

Devolver todas las soluciones

Lo creas o no, esto es **más** sencillo:

```python
def subsetSum_todasSoluciones(valores, objetivo):
  N = len(valores)
  sol, pesoDerecha = [None]*N, [0]*N
  for i in range(N-2,-1,-1):
    pesoDerecha[i] = pesoDerecha[i+1] + valores[i+1]
  def backtracking(longSol, pesoAcum):
    if longSol == N: # hemos llegado a una hoja
      yield (sol,[valores[i] for i in range(N) if sol[i]==1])
    else: # es nodo interno, vamos a RAMIFICAR
      w = valores[longSol]
      if pesoAcum + w <= objetivo: # si no me paso
        sol[longSol] = 1
        yield from backtracking(longSol+1, pesoAcum+w)
      if pesoAcum + pesoDerecha[longSol] >= objetivo:
        sol[longSol] = 0
        yield from backtracking(longSol+1, pesoAcum)
  yield from backtracking(0,0)
```

Devolver todas las soluciones ¿sólo la primera?

- Nada impide, si hemos hecho la versión de todas las soluciones con `yield`, pedirle solamente unas cuantas. Por ejemplo, si queremos pedirle sólo la primera:

```
unasolucion = next(subsetSum_todasSoluciones(valores, objetivo))
print(unasolucion)
```

- La función `next` admite un segundo argumento con un valor por defecto. En este caso vamos a pedir una solución que sume 1000 (lo cual es imposible para los valores del ejemplo):

```
unasolucion = next(subsetSum_todasSoluciones(valores, 10000),
                   "No hay solución")
print(unasolucion)
```

  Mostraría

```
No hay solución
```

Esquema general de *backtracking*

```
def miproblema(datos_entrada):
  N = longitud_solucion
  solucion = [None]*N # para ir dejando la solución
  def factible():
    ...
  def ramificar():
    ...
  def prometedor():
    ...
  def backtracking(longSol): # recibe longitud solución
    if longSol == N: # if es_terminal():
      if factible(): # a veces no hace falta
        yield solucion.copy() # devolvemos una copia
    else: # no es terminal, ramificamos
      for child in ramificar():
        if prometedor():
          solucion[longSol] = child # o lo que haga falta
          yield from backtracking(longSol+1)
  # llamada inicial:
  yield from backtracking(0)
```

Esquema general de *backtracking*

IMPORTANTE:

1. En algunos casos se sabe que una solución prometedora y terminal siempre será factible. En esos casos NO haría falta comprobar factible (esto sirve tanto en la versión `return` como en la versión `yield`):

```
def backtracking(longSol): # recibe longitud solución
    if longSol == N: # if es_terminal():
        # NOS AHORRAMOS if factible():
        return solucion # si paramos no haría falta hacer copia
    ...
```

2. Al comprobar `prometedor` debemos tener en cuenta que el nodo padre, por la forma de funcionar del algoritmo, **también** era prometedor. Por tanto, podemos usar esa información para ahorrarnos muchas comprobaciones.

Actividades

Actividad 1: Permutaciones

Partiendo del código para generar *variaciones con repetición:*

```
def variacionesRepeticion(elementos, cantidad):
    sol = [None]*cantidad
    def backtracking(longSol):
        if longSol == cantidad:
            yield sol.copy()
        else:
            for child in elementos:
                sol[longSol] = child
                yield from backtracking(longSol+1)
    yield from backtracking(0)
```

Se trata de modificarla para generar permutaciones. Para ello debemos evitar ramificar si un elemento ya forma parte de la solución parcial. Es la parte que en el esquema se demonina `prometedor`.

Actividad 1: Permutaciones

El resultado de ejecutar:

```
for x in variacionesRepeticion(['tomate','queso','anchoas'],3):
    print(x)
```

es:

```
['tomate', 'tomate', 'tomate']
['tomate', 'tomate', 'queso']
['tomate', 'tomate', 'anchoas']
['tomate', 'queso', 'tomate']
['tomate', 'queso', 'queso']
['tomate', 'queso', 'anchoas']
['tomate', 'anchoas', 'tomate']
...
['anchoas', 'tomate', 'anchoas']
['anchoas', 'queso', 'tomate']
['anchoas', 'queso', 'queso']
['anchoas', 'queso', 'anchoas']
['anchoas', 'anchoas', 'tomate']
['anchoas', 'anchoas', 'queso']
['anchoas', 'anchoas', 'anchoas']
```

Actividad 1: Permutaciones

El resultado de ejecutar:

```
for x in permutaciones(['tomate','queso','anchoas']):
    print(x)
```

debe ser:

```
['tomate', 'queso', 'anchoas']
['tomate', 'anchoas', 'queso']
['queso', 'tomate', 'anchoas']
['queso', 'anchoas', 'tomate']
['anchoas', 'tomate', 'queso']
['anchoas', 'queso', 'tomate']
```

Actividad 2: Combinaciones

Para generar las combinaciones de un conjunto de elementos *tomados de n en n* debemos crear un vector solución de longitud *n* y en cada vez que ramifiquemos debemos elegir de entre los elementos que sean posteriores al último introducido. Así evitamos que salga varias veces una misma solución. El resultado de ejecutar:

```
for x in combinaciones(['tomate','queso','anchoas','aceitunas'],3):
    print(x)
```

debe ser:

```
['tomate', 'queso', 'anchoas']
['tomate', 'queso', 'aceitunas']
['tomate', 'anchoas', 'aceitunas']
['queso', 'anchoas', 'aceitunas']
```

Actividad 3: Exact cover

El cubrimiento exacto de un conjunto o *exact cover* es un problema muy famoso `https://en.wikipedia.org/wiki/Exact_cover`

- **Entrada:** Dado un conjunto U (de Universo), nos dan un conjunto de subconjuntos de U. Ejemplo:

```
datos = { {1,2,3}, {2,3,4}, {4,5}, {1,5}, {2,3} }
```

- **Objetivo:** Se trata de seleccionar unos cuantos conjuntos de los que nos ha pasado de forma que constituyan una **PARTICIÓN** de U.
  Es decir, que sean disjuntos entre sí y que su unión sea U.
  Para el ejemplo anterior donde `U= {1,2,3,4,5}`:

```
solucion = { {1,2,3}, {4,5} }
```

La solución no tiene por qué existir ni ser única.

En el ejemplo anterior también es solución:

```
solucion = {{2,3,4}, {1,5}}
```

Actividad 3: Exact cover

Representaremos la entrada mediante una **lista** de conjuntos:

```
cjtdcjts = [{"casa","coche","gato"},
            {"casa","bici"},
            {"bici","perro"},
            {"boli","gato"},
            {"coche","gato","bici"},
            {"casa", "moto"},
            {"perro", "boli"},
            {"coche","moto"},
            {"casa"}]
for solucion in exact_cover(cjtdcjts):
    print(solucion)
```

El resultado de ejecutar el código anterior:

```
[{'perro', 'bici'}, {'gato', 'boli'}, {'coche', 'moto'}, {'casa'}]
[{'coche', 'gato', 'bici'}, {'casa', 'moto'}, {'perro', 'boli'}]
```

Actividad 3: Exact cover

Representaremos una solución mediante una secuencia de 0s y 1s donde 1 significa que lo tenemos en cuenta y 0 que no. Pseudocódigo:

```
def exactCover(listaConjuntos):
  U = union listaConjuntos
  N = len(listaConjuntos)
  solucion = []
  def backtracking(longSol, cjtAcumulado):
    if es terminal:
      if es factible:
        yield solucion.copy()
    else: # ramificar
      cjt = listaConjuntos[longSol]
      if cjt y cjtAcumulado son disjuntos:
        añadir cjt en solucion (append)
        yield from backtracking añadiendo cjt a cjtAcumulado
        quitar cjt de solucion (pop)
      # en cualquier caso probar a saltarse cjt
      yield from backtracking sin añadir cjt al cjtAcumulado
  yield from backtracking(0, set()) # empezamos con cjt vacío
```

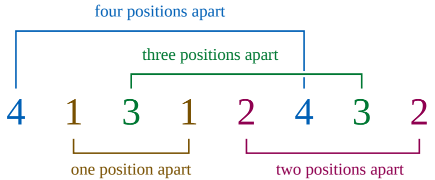
Actividad 3: Exact cover

Completa el siguiente código y comprueba que funciona:

```python
def exact_cover(listaConjuntos, U=None):
    if U is None: # para saber qué universo tenemos
        U = set().union(*listaConjuntos)
    N = len(listaConjuntos)
    solucion = []
    def backtracking(longSol, cjtAcumulado):
        # COMPLETAR
        # consulta los métodos isdisjoint y union de la clase set,
        # podrías necesitarlos
    yield from backtracking(0, set())
```

Actividad 4: Secuencias de Langford

Una secuencia o emparejamiento de Langford de longitud $2N$ es una ordenación de los números $1, 1, 2, 2, \ldots, N-1, N-1, N, N$ de manera que entre dos números i hay i números como en el siguiente ejemplo de wikipedia:



Vamos a utilizar *backtracking*. Utilizaremos un vector (lista Python) de longitud $2N$ que estará inicializada a 0s (valor que denota un hueco en el vector).
El algoritmo de backtracking intentará posicionar, una por una, las N parejas de números empezando por los más altos (la pareja N,N) hasta la más bajita.

Actividad 4: Secuencias de Langford

Ramificar consiste probar todas las posibles posiciones en que podemos situar cada pareja en el vector. Por ejemplo, si vamos a situar el valor 3 debemos tener en cuenta que si el primero ocupa la posición `i` en el vector (con `i` ≥ 0) , el segundo ocupará la posición `i+1+3` (que debe de ser $\leq$ `2*N-1`). Además:

- no podemos poner los valores encima de otros (únicamente podremos situarlos donde previamente tengamos 0s) y que
- al cambiar de rama debemos *deshacer* los cambios efectuados poniendo a 0 las posiciones ocupadas previamente.

Matemáticas al poder

Se puede demostrar que solamente existen soluciones para aquellos valores N tal que el resto de dividir N entre 4 valga 0 o 3. Si no es el caso podemos evitar *backtracking*:

```python
if N%4 in (0,3):
    yield from backtracking(N)
```

Actividad 4: Secuencias de Langford

Un posible esqueleto del programa:

```
def langford(N):
    N2   = 2*N
    seq  = [0]*N2
    def backtracking(num):
        if num<=0:
            yield "-".join(map(str, seq))
        else:
            # buscamos una posicion para situar una pareja num
            pass # COMPLETAR

    if N%4 in (0,3):
        yield from backtracking(N)
```

Actividad opcional

Actividad 5: Langford vía exact cover

- Para reducir el problema de la secuencia de Langford de longitud N al *exact cover* necesitamos *en principio*, un conjunto `U` de longitud $2N$ correspondiente a las posiciones que podemos ocupar en la secuencia. Así, por ejemplo, si el vector se indexa entre `0` y `2*N-1`, podríamos utilizar las cadenas `p0`,`p1`,… como elementos del conjunto `U`.
- Para cada posible número entre 1 y N necesitamos crear un subconjunto por cada posible posición en que podemos meter una pareja de estos valores.

  Ejemplos:
  - para el valor `1` deberíamos considerar los subconjuntos `{'p0','p2'},{'p1','p3'},{'p2','p4'},...`
  - para el valor `2` deberíamos considerar los subconjuntos `{'p0','p3'},{'p1','p4'},{'p2','p5'},...`

  Cada uno de esos conjuntos representa un par de cartas dispuestas a la distancia adecuada.
- Utilizar esto tal cual en *exact cover* no funciona, tiene un fallo ¿cuál?

Actividad 5: Langford vía exact cover

- **¿Cuál es el fallo de este razonamiento?**
  El problema de esta aproximación es que nada impide obtener una solución que utilize más de una vez un mismo número (par de cartas) al tiempo que quedarían números por utilizar.
  Así, por ejemplo, para N=4 en lugar de una secuencia como: `[4,1,3,1,2,4,3,2]` se podría obtener un cubrimiento de `p0,p1,p2,p3,p4,p5,p6,p7` utilizando `{'p0','p2'}`, `{'p1','p3'}`, `{'p4','p6'}`, `{'p5','p7'}` dando la secuencia: `[1,1,1,1,1,1,1,1]`.
- Para resolver el fallo anterior se propone incorporar en el conjunto `U`, además de elementos representando el hecho de ocupar cada una de las `2*N` posiciones del vector, otros `N` valores más que representen el hecho de usar cada uno de los `N` números a utilizar. Estos elementos los denotaremos mediante `n1,n2,...` hasta N.
  Ahora tendremos conjuntos tipo `{'p0','p2','n1'}` para indicar que las 2 cartas con el número `n1` se ha puesto en las posiciones `p0` y `p2`.

Actividad 5: Langford vía exact cover

- Un cubrimiento exacto con conjuntos de esas tripletas va a tener que usar una sola vez cada posible posición y cada posible tipo de carta.
- Completa la siguiente función que recibe `N` y genera la lista de conjuntos para resolver la secuencia de Langford con `exact_cover`:

```python
def langford_data_structure(N):
    # n1,n2,... means that the value has been used
    # p0,p1,... means that the position has been used
    def value(i):
        return sys.intern(f'n{i}')
    def position(i):
        return sys.intern(f'p{i}')
    # crear la lista de conjuntos que resuelva la
    # secuencia de Langford con exact_cover
    return U
```

Internalizar cadenas en Python

`sys.intern(cadena)` permite que al generar varias veces la misma cadena se utilice una sola referencia a una sola copia de la misma.

Actividad 5: Langford vía exact cover

Para terminar, necesitamos una manera de convertir la solución devuelta por el *exact cover* en format de secuencia:

```python
def langford_exact_cover(N):
    if N%4 in (0,3):
        U = langford_data_structure(N)
        sol = [None]*2*N
        for coversol in exact_cover(U):
            for item in coversol:
                elems = sorted(item)
                n = int(elems[0][1:])
                p = int(elems[1][1:])
                sol[p]=n
                p = int(elems[2][1:])
                sol[p]=n
            yield "-".join(map(str,sol))
```