Backtrack en búsquedas combinatorias

Introducción

Lo que hace una búsqueda combinatoria es recorrer todo el espacio de estados de un problema algorítmico de una forma ordenada. Una forma muy común de hacer esto es mediante un procedimiento recursivo llamado **backtrack** (en español **vuelta atrás**), que progresivamente va construyendo un vector $A=(v_1,v_2,\ldots,v_k)$ mediante la adición o eliminación en la parte final de elementos de una lista C de candidatos.

Cada que el algoritmo llega a un vector que podría ser una solución válida, lo procesa.

Para ir recorriendo los vectores, se hace lo siguiente:

```
backtrack(a,k,datos):     si a es vacío:         proceso_inicial(a,k,datos)     si a es valido:         procesar(a,k,datos)
C=generar_candidatos(a,k,datos)     para j en C:         agregar j al final de a         backtrack(a,k,datos)         eliminar j
del final de a     proceso_final(a,k,datos)
```

Como hay diferentes valores de C en distintos valores de la recursión, estos no interfieren entre sí. A grandes rasgos, lo que va pasando es que estamos haciendo una búsqueda a profundidad en el espacio de estados.

Veamos a continuación algunas implementaciones de este estilo que generan permutaciones, conjunto potencia, elecciones y subconjuntos de tamaño k.

Generar permutaciones

```
## Para generar todas las permutaciones de n elementos y procesarlas

def perms(n,a=[],candidates=[],sols=0):
    if len(a)==0:
        candidates=list(range(n))
    if len(a)==n:
        sols+=1
        print(a)
        return(sols)
    for j in candidates:
        a.append(j)
        new_candidates=candidates.copy()
        new_candidates.remove(j)
        sols=perms(n,a,new_candidates,sols)
        a.remove(j)
    return sols

perms(3)
```

```
[0, 1, 2]
[0, 2, 1]
[1, 0, 2]
[1, 2, 0]
[2, 0, 1]
[2, 1, 0]
```

```
6
```

Reto. Utilizar esta idea de backtrack para resolver el problema del reloj que tenemos pendiente.

```
# Esta es una posible implementación que sólo cuenta
# las soluciones y por default lo hace para el reloj,
# aunque optativamente toma un parámetro n para hacerlo para
# menos casos.

def reloj(n=12,a=[],candidates=[],sols=0):
    if len(a)==0:
        candidates=[1]
```

```
    if len(a)==1:
        candidates=list(range(2,n+1))
    if len(a)==2:
        print(a[-1])
    if len(a)>=3:
        if a[-3]+a[-2]+a[-1]==13:
            return sols
        if len(a)==n:
#             print("Good")
            if a[-2]+a[-1]+a[0]==13 or a[-1]+a[0]+a[1]==13:
                return sols
            else:
                return sols+1
    for j in candidates:
        a.append(j)
        new_candidates=candidates.copy()
        new_candidates.remove(j)
        sols=reloj(n,a,new_candidates,sols)
        a.remove(j)
    return sols

print(reloj())
```

```
2
3
4
5
6
7
8
9
10
11
12
24424416
```

Supusimos que la solución comienza con 1, de modo que aún hace falta recuperar las rotaciones de esta solución. En total son $24424416\cdot 12 = 293092992$.

```
print(reloj(11))
```

```
2
3
4
5
6
7
8
9
10
11
1965728
```

Este ya es un muy buen algoritmo pensando en términos de exploración exhaustiva. Toma aproximadamente 1 minuto en terminar. Hay todavía mejores maneras de resolver el problema, explotando mucho más la estructura de cuándo sucede que en efecto tenemos una suma igual a 13 y dando un argumento combinatorio.

Aquí abajo simplemente hay una implementación que además de guardar las cantidad de soluciones, también guarda las soluciones buenas. La ponemos en 6 por default pues para números más grandes hay que tener cuidado con la memoria.

```
def reloj_buenos(n=6,a=[],candidates=[],sols=0,buenos=list([])):
    if len(a)==0:
        candidates=[1]
    if len(a)==1:
        candidates=list(range(2,n+1))
    if len(a)==2:
        print(a[-1])
    if len(a)>=3:
        if a[-3]+a[-2]+a[-1]==13:
            return sols,buenos
        if len(a)==n:
            if a[-2]+a[-1]+a[0]==13 or a[-1]+a[0]+a[1]==13:
                return sols,buenos
            else:
                buenos.append(a.copy())
                return sols+1,buenos
    for j in candidates:
        a.append(j)
        new_candidates=candidates.copy()
        new_candidates.remove(j)
        sols,buenos=reloj_buenos(n,a,new_candidates,sols,buenos)
        a.remove(j)
```

```
        a.remove(j)
    return sols,buenos

reloj_buenos()
```

```
  2
  3
  4
  5
  6
```

```
  (56,
   [[1, 2, 3, 4, 5, 6],
    [1, 2, 3, 5, 4, 6],
    [1, 2, 3, 5, 6, 4],
    [1, 2, 3, 6, 5, 4],
    [1, 2, 4, 3, 5, 6],
    [1, 2, 4, 5, 3, 6],
    [1, 2, 4, 5, 6, 3],
    [1, 2, 4, 6, 5, 3],
    [1, 2, 6, 3, 5, 4],
    [1, 2, 6, 4, 5, 3],
    [1, 3, 2, 4, 5, 6],
    [1, 3, 2, 4, 6, 5],
    [1, 3, 2, 5, 4, 6],
    [1, 3, 2, 6, 4, 5],
    [1, 3, 5, 2, 4, 6],
    [1, 3, 5, 4, 2, 6],
    [1, 3, 5, 4, 6, 2],
    [1, 3, 5, 6, 4, 2],
    [1, 3, 6, 2, 4, 5],
    [1, 3, 6, 5, 4, 2],
    [1, 4, 2, 3, 5, 6],
    [1, 4, 2, 3, 6, 5],
    [1, 4, 2, 5, 3, 6],
    [1, 4, 2, 6, 3, 5],
    [1, 4, 5, 2, 3, 6],
    [1, 4, 5, 3, 2, 6],
    [1, 4, 5, 3, 6, 2],
    [1, 4, 5, 6, 3, 2],
    [1, 4, 6, 2, 3, 5],
    [1, 4, 6, 5, 3, 2],
    [1, 5, 3, 2, 4, 6],
    [1, 5, 3, 2, 6, 4],
    [1, 5, 3, 4, 2, 6],
    [1, 5, 3, 6, 2, 4],
    [1, 5, 4, 2, 3, 6],
    [1, 5, 4, 2, 6, 3],
    [1, 5, 4, 3, 2, 6],
    [1, 5, 4, 6, 2, 3],
    [1, 5, 6, 3, 2, 4],
    [1, 5, 6, 4, 2, 3],
    [1, 6, 2, 3, 4, 5],
    [1, 6, 2, 3, 5, 4],
    [1, 6, 2, 4, 3, 5],
    [1, 6, 2, 4, 5, 3],
    [1, 6, 3, 2, 4, 5],
    [1, 6, 3, 2, 5, 4],
    [1, 6, 3, 5, 2, 4],
    [1, 6, 3, 5, 4, 2],
    [1, 6, 4, 2, 3, 5],
    [1, 6, 4, 2, 5, 3],
    [1, 6, 4, 5, 2, 3],
    [1, 6, 4, 5, 3, 2],
    [1, 6, 5, 3, 2, 4],
    [1, 6, 5, 3, 4, 2],
    [1, 6, 5, 4, 2, 3],
    [1, 6, 5, 4, 3, 2]])
```

Generar elecciones

```
## Para generar elecciones de k elementos distintos de
## entre n, en donde sí importa el orden.

def perms_k(n,k,a=[],candidates=[],sols=0):
    if len(a)==0:
        candidates=list(range(1,n+1))
    if len(a)==k:
        sols+=1
        print(a)
        return(sols)
    for j in candidates:
        a.append(j)
        new_candidates=candidates.copy()
        new_candidates.remove(j)
        sols=perms_k(n,k,a,new_candidates,sols)
        a.remove(j)
    return sols

perms_k(5,3)
```

```
perms_k(5,3)
```

```
[1, 2, 3]
[1, 2, 4]
[1, 2, 5]
[1, 3, 2]
[1, 3, 4]
[1, 3, 5]
[1, 4, 2]
[1, 4, 3]
[1, 4, 5]
[1, 5, 2]
[1, 5, 3]
[1, 5, 4]
[2, 1, 3]
[2, 1, 4]
[2, 1, 5]
[2, 3, 1]
[2, 3, 4]
[2, 3, 5]
[2, 4, 1]
[2, 4, 3]
[2, 4, 5]
[2, 5, 1]
[2, 5, 3]
[2, 5, 4]
[3, 1, 2]
[3, 1, 4]
[3, 1, 5]
[3, 2, 1]
[3, 2, 4]
[3, 2, 5]
[3, 4, 1]
[3, 4, 2]
[3, 4, 5]
[3, 5, 1]
[3, 5, 2]
[3, 5, 4]
[4, 1, 2]
[4, 1, 3]
[4, 1, 5]
[4, 2, 1]
[4, 2, 3]
[4, 2, 5]
[4, 3, 1]
[4, 3, 2]
[4, 3, 5]
[4, 5, 1]
[4, 5, 2]
[4, 5, 3]
[5, 1, 2]
[5, 1, 3]
[5, 1, 4]
[5, 2, 1]
[5, 2, 3]
[5, 2, 4]
[5, 3, 1]
[5, 3, 2]
[5, 3, 4]
[5, 4, 1]
[5, 4, 2]
[5, 4, 3]
```

```
60
```

Generar conjunto potencia

```
## Para generar los subconjuntos de n elementos y procesarlos

def subsets(n,a=[],sols=0):
    print(a)
    sols+=1
    if len(a)==0:
        first=0
    else:
        first=max(a)+1
    candidates=list(range(first,n))
    for j in candidates:
        a.append(j)
        sols=subsets(n,a,sols)
        a.remove(j)
    return sols

subsets(6)
```

```
[]
[0]
```

```
[0, 1]
[0, 1, 2]
[0, 1, 2, 3]
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 5]
[0, 1, 2, 3, 5]
[0, 1, 2, 4]
[0, 1, 2, 4, 5]
[0, 1, 2, 5]
[0, 1, 3]
[0, 1, 3, 4]
[0, 1, 3, 4, 5]
[0, 1, 3, 5]
[0, 1, 4]
[0, 1, 4, 5]
[0, 1, 5]
[0, 2]
[0, 2, 3]
[0, 2, 3, 4]
[0, 2, 3, 4, 5]
[0, 2, 3, 5]
[0, 2, 4]
[0, 2, 4, 5]
[0, 2, 5]
[0, 3]
[0, 3, 4]
[0, 3, 4, 5]
[0, 3, 5]
[0, 4]
[0, 4, 5]
[0, 5]
[1]
[1, 2]
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 4, 5]
[1, 2, 3, 5]
[1, 2, 4]
[1, 2, 4, 5]
[1, 2, 5]
[1, 3]
[1, 3, 4]
[1, 3, 4, 5]
[1, 3, 5]
[1, 4]
[1, 4, 5]
[1, 5]
[2]
[2, 3]
[2, 3, 4]
[2, 3, 4, 5]
[2, 3, 5]
[2, 4]
[2, 4, 5]
[2, 5]
[3]
[3, 4]
[3, 4, 5]
[3, 5]
[4]
[4, 5]
[5]
```


Generar subconjuntos de tamaño \(k\)

```
## Hagamos el procesamiento de soluciones un poco más interesante mediante
## una forma más visual de mostrar los subconjuntos de cierto tamaño.
def display_subset(n,S):
    pic=n*["_"]
    for j in S:
        pic[j]="O"
    pic=str(S)+': '+''.join(pic)
    print(pic)

def subsets_k(n,k,a=[],sols=0):
    if len(a)==k:
        display_subset(n,a)
        sols+=1
    if len(a)==0:
        first=0
    else:
        first=max(a)+1
    candidates=list(range(first,n))
    for j in candidates:
        a.append(j)
        sols=subsets_k(n,k,a,sols)
        a.remove(j)
    return sols

subsets_k(7,3)
```

```
subsets_k(7,3)

[0, 1, 2]: OOO____
[0, 1, 3]: OO_O___
[0, 1, 4]: OO__O__
[0, 1, 5]: OO___O_
[0, 1, 6]: OO____O
[0, 2, 3]: O_OO___
[0, 2, 4]: O_O_O__
[0, 2, 5]: O_O__O_
[0, 2, 6]: O_O___O
[0, 3, 4]: O__OO__
[0, 3, 5]: O__O_O_
[0, 3, 6]: O__O__O
[0, 4, 5]: O___OO_
[0, 4, 6]: O___O_O
[0, 5, 6]: O____OO
[1, 2, 3]: _OOO___
[1, 2, 4]: _OO_O__
[1, 2, 5]: _OO__O_
[1, 2, 6]: _OO___O
[1, 3, 4]: _O_OO__
[1, 3, 5]: _O_O_O_
[1, 3, 6]: _O_O__O
[1, 4, 5]: _O__OO_
[1, 4, 6]: _O__O_O
[1, 5, 6]: _O___OO
[2, 3, 4]: __OOO__
[2, 3, 5]: __OO_O_
[2, 3, 6]: __OO__O
[2, 4, 5]: __O_OO_
[2, 4, 6]: __O_O_O
[2, 5, 6]: __O__OO
[3, 4, 5]: ___OOO_
[3, 4, 6]: ___OO_O
[3, 5, 6]: ___O_OO
[4, 5, 6]: ____OOO

35
```

Tarea moral

Los siguientes problemas te ayudarán a practicar lo visto en esta entrada. Para resolverlos, necesitarás usar herramientas matemáticas, computacionales o ambas.

1. Problema
2. Problema
3. Problema
4. Problema
5. Problema

Anterior
Recursión y teorema maestro

Siguiente
Más ejemplos de backtrack

Por Leonardo Ignacio Martínez Sandoval