Recortes al espacio de estados

Introducción

Exploración exhaustiva recortada

Consiste en dejar de explorar posibilidades que ya no se pueden extender a posibilidades exitosas. Esto no es lo mismo que reducir el espacio de estados. Tampoco quiere decir que no exploremos todas las posibilidades. Consiste en dar argumentos para que nuestro algoritmo revise menos casos y siga dando la respuesta correcta.

Retomemos uno de los problemas de la sección anterior:

Problema 1. ¿De cuántas formas se puede poner a \(10000\) como suma de cuadrados de dos números enteros positivos? ¿En cuál de las expresiones \(x^2+y^2=10000=100^2\) se minimiza \(3x+5y-1\)?

Recortemos el espacio de estados a partir de las siguientes dos observaciones:

- Ambos \(x\) y \(y\) tienen que ser pares
- Si \(x\) ya está dada, no tiene chiste mover \(y\) por todos sus valores posibles de \(1\) a \(100\) pues por tamaño ya sólo puede tener pocas posibilidades.

\(y\) ya nada más puede ser a lo mucho \(\approx\sqrt{10000-x^2}\). Esto ahorra algunos pasos.

Los otros problemas también se prestan a este tipo de recorte de espacio de estados.

```
soluciones=[]
for x in range(1,100):
    for y in range(1,int((10000-x**2)**(0.5)+1)):
        if x**2+y**2==10000:
            soluciones.append((x,y))

print(soluciones)
```

```
[(28, 96), (60, 80), (80, 60), (96, 28)]
```

Problemas de exploración exhaustiva recortada

Pongamos otros cuantos problemas que se pueden estudiar usando un recorte de espacio de estados.

Problema 6. Tenemos que encontrar todas las parejas de palabras `x` y `y` en español que sean diferentes y que `x y` tenga en total 9 caracteres.

Problema 7. Queremos encontrar para cuantas parejas `x` y `y` de palabras en español sucede que cada una de ellas tiene por lo menos cinco letras y las últimas cinco letras de la primera son iguales a las primeras cinco letras de la última.

Problema 8. Tenemos el siguiente arreglo de números

\[[4,1,7,4,2,5,5,7,1,8,4,9,9,1,4,1,5,7,2,8,3,6,1]\]
. Queremos determinar de cuántas formas se pueden elegir algunos de estos números de forma consecutiva de modo que sumen \(18\).

Problema 9. Una **matriz mágica sencilla** consiste de una matriz de \(3\times 3\), la suma de las entradas en cada fila es un cierto número \(x\) y la suma de las entradas en cada columna es ese mismo número \(x\). Las entradas deben ser los números del \(1\) al \(9\). ¿Cuántas matrices mágicas existen?

(9\). ¿Cuántas matrices mágicas existen?

Problema 10. Se cayeron los \(12\) números de un reloj. Se quieren volver a poner, uno en cada posición. No importa tanto saber la hora, pero es muy importante que la suma de tres de esos números consecutivos (en orden cíclico) no sea \(13\), porque da mala suerte. ¿De cuántas formas es posible hacer esto?

Veamos los problemas uno por uno.

Problema 6. Tenemos que encontrar todas las parejas de palabras `x` y `y` en español que sean diferentes y que `x y` tenga en total 9 caracteres.

Podemos pensar el espacio de estados como todas las parejas de palabras en español y procesar cada una de ellas. Si lo exploramos de manera directa, esto toma tiempo cuadrático en la cantidad de palabras en español. Al procesar cada posibilidad seguro que cubrimos todas, pero parece que hay cierta pérdida de tiempo pues las palabras grandes (de \(7\) letras o más) las estamos considerando en muchas parejas, pero todas ellas van a fallar.

Una mejor forma de explorar el espacio de estados es primero usar tiempo lineal en descartar las palabras grandes y luego usar tiempo cuadrático en una lista mucho más corta.

```
lista=open('espanol.txt','r',encoding = "ISO-8859-1")
linea=lista.readline()

# Vamos a encontrar todas las palabras que sean cortas y guardarlas en esta lista.
cortas=[]
while linea:
    # Algunas líneas de nuestra lista incluyen 'subj' al final, por eso primero
    # las limpiamos con esto.
    limpio=linea[:-1].split(' ')[0]
    # Ahora sí, procesamos la cadena limpia. Típico algoritmo que va almacenando
    # las que cumplen.
    if len(limpio)<=7:
        cortas.append(limpio)
    linea=lista.readline()

print(len(cortas))
print("Los casos a checar sin cortar la búsqueda son aproximadamente " + str(170000**2))
print("Los casos en la búsqueda cortada son aproximadamente " + str(32000**2))
```

```
32971
Los casos a checar sin cortar la búsqueda son aproximadamente 28900000000
Los casos en la búsqueda cortada son aproximadamente 1024000000
```

Ya tenemos la lista de palabras cortas. Con esta lista, ahora sí podríamos hacer una exploración exhaustiva. Como hay como 1024 millones de casos a verificar, esto tardaría ~1024 segundos en terminar. ¿Cómo podemos seguir reduciendo el espacio de estados? Agreguemos una idea de ordenar la lista de palabras por su longitud. Después, recortaremos la búsqueda. En cuanto la palabra del bucle interno es demasiado grande, se cierra esa ejecución del bucle externo. Como la lista está ordenada por longitud, no hay soluciones que se pierdan.

Notemos lo que estamos haciendo:

- Estamos usando tiempo \(O(n)\) primero para reducir la lista a una de palabras más cortas.
- Estamos usando tiempo \(O(n\log n)\) en la lista de palabras cortas para ordenarla.
- Así, el tiempo \(O(n^2)\) que usamos lo podemos recortar. Esto es muy conveniente pues es lo que asintóticamente tiene la carga más pesada.

```
soluciones=0
cortas=sorted(cortas, key=lambda x: len(x))
print(cortas[:70])

# A continuación se muestra una búsqueda recortada, que en cuanto una candidata a segunda palabra
# encuentra una palabra demasiado grande, rompe la búsqueda y no hace ya nada más para la primer
# palabra en curso.
for j in cortas:
    for k in cortas:
        if len(j) + len(k) == 8:
            soluciones+=1
        # Aquí cortamos la exploración. Hay que notar que es bien importante que la lista de palabras
        # cortas está ordenada, pues si no correríamos el riesgo de perder soluciones al hacer esto.
```

```
        if len(j) + len(k) >8:
            break

print(soluciones)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y',
7099844
```

Acá arriba recortamos la búsqueda en cuanto estuvimos seguros de que ya no habría soluciones. Esto nos dejó todavía con un algoritmo en $O(n^2)$, pero, con un mejor factor constante que permitió correrlo en pocos segundos. Esto todavía no es el mejor algorimo, pero refleja bien la idea de recorte de espacio de estados.

Los siguientes dos problemas quedan como práctica.

Problema 7. Queremos encontrar para cuantas parejas de `x` y `y` de palabras en español sucede que cada una de ellas tiene por lo menos cinco letras y las últimas cinco letras de la primera son iguales a las primeras cinco letras de la última.

Problema 8. Tenemos el siguiente arreglo de números

$$[[4,1,7,4,2,5,5,7,1,8,4,9,9,1,4,1,5,7,2,8,3,6,1]$$
. Queremos determinar de cuántas formas se pueden elegir algunos de estos números de forma consecutiva de modo que sumen 18.

Problema 9. Una **matriz mágica sencilla** consiste de una matriz de 3×3, la suma de las entradas en cada fila es un cierto número x y la suma de las entradas en cada columna es ese mismo número x. Las entradas deben ser los números del 1 al 9, sin repetir ¿Cuántas matrices mágicas existen?

Lo primero que tenemos que hacer es decidir quién será nuestro espacio de estados. Como los números no se repiten, podemos pensar en permutaciones. Esto en total nos da un espacio de estados con 9! elementos a verificar. Para hacer esto, tomamos la función auxiliar de permutaciones que hicimos arriba.

La permutación $[a_0,\ldots,a_8]$ la pensaremos como la matriz

$$\begin{pmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{pmatrix}$$
Una observación adicional es que la suma de todos los números del 1 al 9 es 45. De este modo, la suma en cada fila y en cada columna debe ser igual a 15. Esto nos permite comparar la suma de cada fila y columna no entre sí, sino entre ellas y un número constante. Esto nos permite poner la condición de que la matriz sea mágica a ciertas sumas igualadas a 15.

```
numeros=[1,2,3,4,5,6,7,8,9]
candidatas=perms(numeros)
print(len(permutaciones))

# La siguiente función recibe un vector de $9$ entradas que corresponden a una matriz
# y responde de manera booleana si la matriz es mágica o no, de acuerdo a la definición
# del problema.
def es_magica_3(lista):
    a=(lista[0]+lista[3]+lista[6]==15)
    b=(lista[1]+lista[4]+lista[7]==15)
    c=(lista[2]+lista[5]+lista[8]==15)
    d=(lista[0]+lista[1]+lista[2]==15)
    e=(lista[3]+lista[4]+lista[5]==15)
    f=(lista[6]+lista[7]+lista[8]==15)
    return a and b and c and d and e and f
```

```
3628800
```

```
soluciones=[]
# Típica forma de ir almacenando las soluciones
for matriz in candidatas:
    if es_magica_3(matriz):
        soluciones.append(matriz)

print(len(soluciones))
print(soluciones[0])
```

```
72
[9, 4, 2, 1, 8, 6, 5, 3, 7]
```

```
#¿Qué tal que queremos ver la matriz un poco más linda?

for k in range(3):
    print(soluciones[0][3*k:3*(k+1)])
```

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

Esta solución es suficientemente rápida para matrices de 3×3, pero es muy lenta si quisiéramos encontrar matrices mágicas más grandes. Cuando tenemos matrices de 4×4, la cantidad de números que queremos acomodar es $16!\approx 10^{13}$ (como 10 millones de segundos) y eso es demasiado grande para esperar. ¿Cómo podemos cambiar o recortar el espacio de estados en este caso?

Primero generalicemos algunas de las observaciones anteriores. En una matriz mágica de $n\times n$ tenemos los números de 1 a n^2 y por lo tanto la suma de todos ellos es

$$1+2+\ldots+n^2=n^2(n^2+1)/2,$$
de donde la suma en cada renglón y columna debe ser igual a $n(n^2+1)/2$. Para $n=4$ obtenemos 34.

```
# La siguiente función recibe un vector de $16$ entradas que corresponden a una matriz
# y responde de manera booleana si la matriz es mágica o no, de acuerdo a la definición
# del problema.
def es_magica_4(lista):
    magica=True
    # Verificar que las filas sumen 34.
    for fila in range(4):
        magica=magica and (sum(lista[entrada] for entrada in range(4*fila,4*fila+4))==34)
    # Verificar que las columnas sumen 34.
    for columna in range(4):
        magica=magica and (sum(lista[entrada] for entrada in range(columna,16,4))==34)
    return magica

#Hacemos una prueba con una matriz que sepamos que es mágica
A=[8,11,14,1,13,2,7,12,3,16,9,6,10,5,4,15]
print(es_magica_4(A))
#Y con una que no
B=[8,11,1,14,13,2,7,12,3,16,9,6,10,5,4,15]
print(es_magica_4(B))
```

```
True
False
```

Retomaremos este problema más adelante, cuando hablemos de bactrack y búsquedas combinatorias.

Recortes de espacio de estados con simetrías

Es usar simetrías en el espacios de estados y el problema planteado para reducir la exploración. Por ejemplo, si estamos buscando cuántas parejas (x,y) de una lista de núméros suman cierto número M, entonces una posibilidad es que recorramos todo el espacio de estados para x y y, pero otra forma de hacerlo es suponer que $x\leq y$ y luego tras encontrar todas las soluciones bajo esta suposición, simplemente a partir de cada pareja (x,y) construir la pareja (y,x).

Apliquemos esta esta idea en siguiente problema.

Problema 10. Se cayeron los 12 números de un reloj. Se quieren volver a poner, uno en cada posición. No importa tanto saber la hora, pero es muy importante que la suma de tres de esos números consecutivos (en orden cíclico) no sea 13, porque da mala suerte. ¿De cuántas formas es posible hacer esto?

Si hiciéramos una solución que explore todos los posibles estados, entonces necesitaríamos pasar por $12!$ de ellos, que son como 480 millones. Sin embargo, podemos ahorrarnos un factor de 12 si observamos que tenemos una simetría rotacional de orden 12:

un acomodo funciona si y sólo si funcionan todas sus rotaciones. Por esta razón, basta con considerar aquellas permutaciones que comiencen con \(1\) y verificar lo que queremos.

```
numeros=[2,3,4,5,6,7,8,9,10]
candidatas=[[1]+j for j in perms(numeros)]
print(len(permutaciones))
```

```
3628800
```

```
candidatas[:5]
```

```
[[1, 10, 9, 8, 7, 6, 5, 4, 3, 2],
 [1, 10, 9, 8, 7, 6, 5, 4, 2, 3],
 [1, 10, 9, 8, 7, 6, 5, 3, 4, 2],
 [1, 10, 9, 8, 7, 6, 5, 2, 4, 3],
 [1, 10, 9, 8, 7, 6, 5, 3, 2, 4]]
```

```
def buena_suerte(lista):
    buena=True
    for j in range(10):
        buena = buena and (lista[(j%10)]+lista[(j+1)%10]+lista[(j+2)%10]!=13)
    return buena

soluciones=0
for lista in candidatas:
    if buena_suerte(lista):
        soluciones+=1

print(soluciones)
```

```
165520
```

Tenemos 165520 soluciones que comienzan con \(1\). Si queremos regresar al problema con 10 números, todavía cada una de estas soluciones crea 10 soluciones, una por cada una de las 10 rotaciones posibles. Así, la cantidad total de soluciones es 1655200.

Introducción

Sección 1

Sección 2

Sección 3

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

Por Leonardo Ignacio Martínez Sandoval