informe

June 4, 2023

1 Trabajo Práctico 2 - Problema de Empaquetamiento

Curso Buchwald & Genender - 20231C

Alumnos: - Felipe de Luca Andrea - 105646 - Francisco de Luca Andrea - 109794

1.1 Definición del problema

Dado un conjunto de n objetos cuyos tamaños son $\{T_1, T_2, \cdots, T_n\}$, con $T_i \in (0, 1]$, se debe empaquetarlos usando la mínima cantidad de envases de capacidad 1.

1.2 Resolución

1.2.1 1) Demostrar que el problema de empaquetamiento es NP-Completo.

El problema de decisión de este problema es el siguiente:

> Dados n objetos cuyos tamaños son $\{T_1, T_2, \cdots, T_n\}$, con $T_i \in (0, 1]$, y un número k, ¿es posible empaquetarlos en como mucho k envases de capacidad 1?

1. **El problema es NP**

   Dados:

   - Un conjunto de objetos: $T = \{T_1, T_2, \cdots, T_n\}$, con $T_i \in (0, 1]$
   - Una solución al problema, es decir, un conjunto de envases con los objetos de T: $E = \{E_1, E_2, \cdots, E_k\}$
   - La cantidad de envases: $k = |E|$

   Debemos demostrar que existe un algoritmo polinomial que permita verificar que la soluición E es válida. Es sencillo ver que este algoritmo es polinomial: basta con recorrer todos los envases en E verificando que la suma de los tamaños de los objetos en cada envase sea menor o igual a 1, y que al terminar de recorrer todos los envases, todos los objetos hayan sido empaquetados en exactamente un envase. Esto se puede hacer en tiempo $O(n)$ con el siguiente algoritmo:

```
restantes = set(T)
for cada envase en E:
    suma = 0
    for cada objeto en envase:
        suma += tamaño del objeto
        if objeto no está en restantes:
            return False
        restantes.remove(objeto)
```

```
    if suma > 1:
        return False
return len(restantes) == 0
```

2. **El problema es NP-Completo**

   Habiéndo demostrado que el problema es NP, podemos demostrar que es NP-completo reduciéndo otro problema NP-Completo a este. Para ello, vamos a utilizar el problema de *Balanceo de carga*. Este problema es NP-hard [1] y su problema de decisión es NP-completo [2]. El problema de decisión es:

   > Dadas m' máquinas y n' trabajos, y cada trabajo toma tiempo $T'_j \in T'$, ¿es posible asignar los trabajos a las máquinas de forma que el tiempo de ejecución de cada máquina sea menor o igual a k'?

   Podemos reducirlo polinoimalmente al problema de empaquetamiento. Notar que, para cualquier conjunto de trabajos $A \subset T'$:

   $$\sum_{t \in A} t \leq k' \Leftrightarrow \sum_{t \in A} \frac{t}{k'} \leq 1$$

   Si evaluamos el problema de empaquetamiento donde los tamaños de cada objeto son $T_j = \frac{T'_j}{k'}$ en a lo sumo m' envases, tenemos entonces que:

   1. Si el problema de empaquetamiento tiene solución, significa que fue posible asignar los objetos en m' envases de forma tal que la suma en cada uno sea a lo sumo 1. Por lo tanto, si tomamos los trabajos de cada envase y los asignamos a una máquina, el tiempo de ejecución de cada máquina será menor o igual a k', dando como resultado una solución válida del problema de balanceo de carga.

      $$\text{Empaquetamiento} \Rightarrow \text{Balanceo de carga}$$

   2. De la misma manera, si el problema de balanceo de carga tiene una solución, entonces el problema de empaquetamiento va a tenerla también: se empaquetan los trabajos de cada máquina en un envase, que como sus tiempos suman a lo sumo k', entonces los tamaños de cada envase no superarán 1.

      $$\text{Balanceo de carga} \Rightarrow \text{Empaquetamiento}$$

   Los casos donde no hay solución son sus contrarecíprocos.

   $$\therefore \text{ Empaquetamiento} \Leftrightarrow \text{Balanceo de carga}$$

---

1. De acuerdo a las diapositivas de la cátedra
2. El problema de decisión de balanceo de carga $D(k')$ es NP-completo ya que:
   1. El problema de optimización es NP-hard
   2. El problema de decisión es NP: se puede implementar un algoritmo de verificación polinomial. Es, de hecho, el mismo algoritmo que para empaquetamiento solo que recorriendo trabajos en máquinas en vez de objetos en envases, y verificando que la suma sea menor a k' en vez de 1

1.2.2 2) Programar un algoritmo por Backtracking/Fuerza Bruta que busque la solución exacta del problema. Indicar la complejidad del mismo. Realizar mediciones del tiempo de ejecución, y realizar gráficos en función de n.

El algoritmo se encuentra en el archivo algoritmos/backtracking.py. A continuación un ejemplo de su ejecución:

[1]:
```
import numpy as np
from algoritmos.backtracking import backtracking
np.random.seed(99)

# El 1-x es para pasar de [0, 1) a (0, 1]
objects =  [round(1 - x, 3) for x in np.random.random(16)]
print(f"Tamaños: {objects}\n")
solution = backtracking(objects, debug=True)
print(f"\nMejor solución: {len(solution)} envases\n{solution}")
```

```
Tamaños: [0.328, 0.512, 0.175, 0.969, 0.192, 0.434, 0.702, 0.953, 0.009, 0.993,
0.23, 0.253, 0.623, 0.506, 0.071, 0.605]

Lower bound: 8
Previous best: 16, new best: 9
[[0.328, 0.512, 0.009, 0.071], [0.175, 0.192, 0.434], [0.969], [0.702, 0.23],
[0.953], [0.993], [0.253, 0.623], [0.506], [0.605]]
Previous best: 9, new best: 8
[[0.328, 0.512, 0.009, 0.071], [0.175, 0.192, 0.623], [0.969], [0.434, 0.506],
[0.702, 0.23], [0.953], [0.993], [0.253, 0.605]]

Mejor solución: 8 envases
[[0.328, 0.512, 0.009, 0.071], [0.175, 0.192, 0.623], [0.969], [0.434, 0.506],
[0.702, 0.23], [0.953], [0.993], [0.253, 0.605]]
```

Explicación del algoritmo

Se implementó un algoritmo por backtracking recursivo. En cada invocación recursiva, el algoritmo coloca el siguiente objeto en un envase. Para cada objeto, hay 2 opciones:

1. Colocar el objeto en uno de los envases preexistentes
2. Colocar el objeto en un nuevo envase

De esta manera, se generan todas las combinaciones posibles de objetos en envases sin repetir soluciones. Cuando se recorrieron todos los objetos, se llegó a una solución válida y se devuelve la cantidad de paquetes.

Podas implementadas

- Si ya se estan usando la misma o más cantidad de paquetes que la mejor solución encontrada hasta el momento
- Si la suma de los objetos en un envase supera 1 (directamente no se agrega el objeto en ese envase)

- Si mejor solución encontrada es igual a la cota inferior:

$$z(I) \geq \left\lceil \sum_{t \in T} t \right\rceil$$

Complejidad computacional

Una forma sencilla de acotar la complejidad del algoritmo es considerar que la cantidad de paquetes nunca va a ser mayor a la cantidad de objetos. Si consideramos que tenemos n objetos para poner en a lo sumo n envases, entonces la complejidad del algoritmo es $O(n^n)$: cada uno de los objetos en cualquiera de los (a lo sumo) n envases.

Se puede encontrar una mejor cota considerando que la cantidad de paquetes nunca va a ser mayor a la cantidad de objetos colocados hasta el momento. Por como funciona el algoritmo, el primer objeto siempre se va a colocar en un paquete nuevo, el segundo objeto siempre se va a colocar en un paquete nuevo o en el mismo paquete que el primero, y así sucesivamente. De esta manera procesar el primer objeto requiere 1 operación, procesar el segundo objeto requiere a lo sumo 2 operaciones, procesar el tercer objeto requiere a lo sumo 3 operaciones, y así sucesivamente. Por lo tanto, la complejidad del algoritmo queda acotada por $O(n!)$.

Podemos acotarlo aun mejor. Defino el tiempo que toma el algoritmo como $T(n) = T_n(0,0)$, donde T_n es la el tiempo del algoritmo recursivo dada la cantidad de objetos colocados hasta el momento $i \in \mathbb{Z}, 0 < i < n$, y la cantidad de envases utilizados hasta el momento (menos 1) $j \in \mathbb{Z}, 0 < j < n$. La ecuación de recurrencia del algoritmo T_n (sin considerar las podas) es:

$$\begin{cases} T_n(n,j) = O(1) \\ T_n(i,j) = j \times T(i+1,j) + T(i+1,j+1), \text{ si } i < n \end{cases}$$

El término de $j \times T(i+1,j)$ representa los casos donde se intenta poner el objeto en uno de los envases existentes, y el término $T(i+1,j+1)$ representa el caso donde se intenta poner el objeto en un nuevo envase. Cuando ya se colocaron todos los objetos ($i = n$), el algoritmo termina por lo que $T_n(n,j) = O(1)$.

Definiendo $T'(i,j) = T_n(n-i,j)$, tenemos que $T(n) = T'(n,0)$ y podemos reescribir la ecuación de recurrencia como:

$$\begin{cases} T_n(0,j) = O(1) \\ T_n(i,j) = j \times T(i-1,j) + T(i-1,j+1), \text{ si } i > 0 \end{cases}$$

Esta ecuación de recurrencia $T'(i,j)$ da como resultado lo que se conoce como *número de r-Bell* [3] donde $T'(i,j) = O(B_{n,r})$. De esta manera tenemos que el algoritmo de backtracking esta acotado por $T(n) = T'(n,0) = O(B_{n,0}) = O(B_n)$, el enésimo número de Bell. Los números de Bell también se definen con otra ecuación de recurrencia más conocida, $B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$. El enésimo número de Bell es el número de particiones de un conjunto de n elementos, es decir, cuantas formas hay de dividir un conjunto de n elementos en subconjuntos no vacíos: es exáctamente lo que hace el algoritmo de backtracking, probar todas las formas de dividir los objetos en envases. Es considerablemente menor que $n!$, y está acotado [4] por $B_n < \left(\frac{0.792n}{ln(n+1)}\right)^n$.

Las mediciones y gráficos se realizan al final del informe.

3. Mezö, I. (2011). "The r-Bell Numbers". *Journal of Integer Sequences*, **14** (11.1.1): pp. 1-14.

   En este paper se menciona la definición de los números de r-Bell y su relación con los números de Bell. en la página 12 se prueba la equivalencia de la definición de los números de r-Bell dada por el paper con la ecuación de recurrencia encontrada para el algoritmo.

4. Berend, D.; Tassa, T. (2010). "Improved bounds on Bell numbers and on moments of sums of random variables". *Probability and Mathematical Statistics*, **30** (2): 185–205.

1.2.3 3) Considerar el siguiente algoritmo: Se abre el primer envase y se empaqueta el primer objeto, luego por cada uno de los objetos restantes se prueba si cabe en el envase actual que está abierto. Si es así, se lo agrega a dicho envase, y se sigue con el siguiente objeto. Si no entra, se cierra el envase actual, se abre uno nuevo que pasa a ser el envase actual, se empaqueta el objeto y se prosigue con el siguiente. Este algoritmo sirve como una aproximación para resolver el problema de empaquetamiento. Implementar dicho algoritmo, analizar su complejidad, y analizar cuán buena aproximación es. Para esto, considerar lo siguiente: Sea I una instancia cualquiera del problema de empaquetamiento, y $z(I)$ una solución óptima para dicha instancia, y sea $A(I)$ la solución aproximada, se define $\frac{A(I)}{z(I)} \leq r(A)$ para todas las instancias posibles. Calcular $r(A)$ para el algoritmo dado, demostrando que la cota está bien calculada. Realizar mediciones utilizando el algoritmo exacto y la aproximación, con el objetivo de verificar dicha relación.

El algoritmo se encuentra en el archivo `algoritmos/greedy_consigna.py`. Tiene complejidad $O(n)$ en tiempo. A continuación un ejemplo de su ejecución:

[2]:
```
import numpy as np
from algoritmos.greedy_consigna import greedy_consigna
np.random.seed(99)

objects =  [round(1 - x, 3) for x in np.random.random(16)]
print(f"Tamaños: {objects}")
solution = greedy_consigna(objects)
print(f"\nSolución: {len(solution)} envases\n{solution}")
```

```
Tamaños: [0.328, 0.512, 0.175, 0.969, 0.192, 0.434, 0.702, 0.953, 0.009, 0.993,
0.23, 0.253, 0.623, 0.506, 0.071, 0.605]

Solución: 11 envases
[[0.328, 0.512], [0.175], [0.969], [0.192, 0.434], [0.702], [0.953, 0.009],
[0.993], [0.23, 0.253], [0.623], [0.506, 0.071], [0.605]]
```

Dados: - Un conjunto de objetos: $T = \{T_1, T_2, \cdots, T_n\}$, con $T_i \in (0, 1]$ - Una solución al problema, es decir, un conjunto de envases con los objetos de T: $E = \{E_1, E_2, \cdots, E_k\}$ - La cantidad de envases: $k = |E|$

La solución óptima sería $z(I) = k$ mínimo, y esta acotada por $z(I) \geq \left\lceil \sum_{t \in T} t \right\rceil \geq \sum_{t \in T} t$, ya que a lo sumo podemos llenar perfectamente $k - 1$ envases y poner los objetos restantes en el último.

También sabemos que $z(I) \leq n$, ya que la peor solución sería cada objeto en un envase distinto.

El peor caso de esta aproximación sería cuando no pone un objeto en un envase anterior porque no entra (cualquier otro caso, mejora la solución de la aproximación sin que necesariamente mejore la óptima). Si pasa para todos los objetos, daría como resultado una solución $A(I) = n$. Para ello, se tiene que dar que $T_i + T_{i+1} > 1$ para todos los objetos consecutivos. Entonces:

$$z(I) \geq \sum_{t \in T} t = T_1 + T_2 + T_3 + T_4 + \cdots + T_n > 1 + 1 + \cdots = \begin{cases} \frac{n}{2} & \text{si n es par} \\ \frac{n-1}{2} + T_n & \text{si n es impar} \end{cases}$$

Notar que para n impar, $z(I) > \frac{n-1}{2} + T_n > \frac{n-1}{2}$. Nunca va a valer exactamente $\frac{n-1}{2}$ por la condición de $T_i + T_{i+1} > 1$ y también porque $T_i > 0$. Como para n impar $\frac{n-1}{2}$ es un entero, tenemos que el mínimo $z(I)$ es el entero siguiente: $z(I) \geq \frac{n-1}{2} + 1 = \frac{n+1}{2} > \frac{n}{2}$. Por lo tanto, vale la misma desigualdad $z(I) > \frac{n}{2}$ para n par e impar.

$$\Rightarrow z(I) > \frac{n}{2} \Rightarrow \frac{1}{z(I)} < \frac{2}{n} \Rightarrow \frac{A(I)}{z(I)} < n \times \frac{2}{n} = 2$$
$$\therefore r(A) \leq 2$$

Las mediciones y gráficos se realizan al final del informe.

1.2.4 4) [Opcional] Implementar alguna otra aproximación (u algoritmo greedy) que les parezca de interés. Comparar sus resultados con los dados por la aproximación del punto 3. Indicar y justificar su complejidad.

El algoritmo propuesto ordena los objetos de mayor a menor y coloca cada objeto en el envase más vacío. Si no entra en ninguno, crea un nuevo envase. La complejidad del algoritmo es $O(n \log n)$ en tiempo, ya que ordena los objetos y luego los recorre una vez realizando operaciones constantes y logarítmicas. Por cada vez que recorre un objeto, busca el envase más vacío. Como son como mucho n envases, esto se hace en $O(log(n))$ utilizando un heap.

Se encuentra en el archivo `algoritmos/greedy_alternativo.py`. A continuación un ejemplo de su ejecución:

[3]:
```
import numpy as np
from algoritmos.greedy_alternativo import greedy_alternativo
np.random.seed(99)

objects =  [round(1 - x, 3) for x in np.random.random(16)]
print(f"Tamaños: {objects}")
solution = greedy_alternativo(objects)
print(f"\nSolución: {len(solution)} envases\n{solution}")
```

```
Tamaños: [0.328, 0.512, 0.175, 0.969, 0.192, 0.434, 0.702, 0.953, 0.009, 0.993,
0.23, 0.253, 0.623, 0.506, 0.071, 0.605]

Solución: 9 envases
[[0.175, 0.071, 0.009], [0.512, 0.328], [0.605, 0.253], [0.623, 0.23], [0.953],
[0.969], [0.702, 0.192], [0.993], [0.506, 0.434]]
```

1.2.5 Comparación de los algoritmos

Para comparar los algoritmos, hicimos pruebas guardando la solución obtenida y el tiempo empleado para cada algoritmo. A continuación se definen los parámetros utilizados para dichas pruebas.

[4]:
```
N_MIN = 2   # Mínima cantidad de objetos con los que probar los algoritmos
N_MAX = 25  # Máxima cantidad de objetos con los que probar los algoritmos
S = 10      # Cantidad de muestras para cada N
P = 15      # Cantidad de puntos entre 0 y N a probar
```

Código para medición de los tiempos:

[5]:
```
from typing import Callable, TypeVar
from time import time
import numpy as np
from algoritmos import Algorithm
from algoritmos.backtracking import backtracking

T = TypeVar("T")
def time_func(func: Callable[[], T]) -> tuple[float, T]:
    """
    Devuelve el tiempo que tarda en ejecutarse la función y su resultado
    """
    start = time()
    ret = func()
    return time() - start, ret

# Algoritmo -> n objetos -> [V resultados]
Data = dict[Algorithm, dict[int, list[int | float]]]

x = np.linspace(N_MIN, N_MAX, P, dtype=int)
# Resultados obtenidos
solutions: Data = {a: {i: [None]*S for i in x} for a in Algorithm}
# Tiempos medidos
times: Data = {a: {i: [None]*S for i in x} for a in Algorithm}

# Devuelve (n, sample, Algoritmo -> (solución, tiempo))
def run(n: int, sample: int) -> tuple[int, int, dict[Algorithm, tuple[int,␣
 ↪float]]]:
    """
    Ejecuta los algoritmos con n objetos y devuelve los resultados
    """
    np.random.seed(n*123456 + sample)

    objects = list(1 - np.random.random(n))
    results = {}
    best_approx = None
    for al in (Algorithm.GREEDY, Algorithm.GREEDY_ALT):
```

```
        t, solution = time_func(lambda: al.func()(objects))
        results[al] = (len(solution), t)
        if best_approx is None or len(solution) < len(best_approx):
            best_approx = solution

    t, solution = time_func(lambda: backtracking(objects,␣
↪better_than=best_approx))
    results[Algorithm.EXACT] = (len(solution), t)
    return n, sample, results
```

[6]:
```
from concurrent.futures import ProcessPoolExecutor, as_completed

def log(completed, n, s):
    print(" " * 100, end="\r")
    print(f"Timing algorithms: {100*completed/(S*P) :.2f}%. Last completed:␣
↪[n={n}, sample={s}]", end="\r")

futures = []
with ProcessPoolExecutor() as p:
    for n in x:
        for s in range(S):
            futures.append(p.submit(run, n, s))
    completed = 0
    for r in as_completed(futures):
        n, s, results = r.result()
        completed += 1
        for a, (solution, time) in results.items():
            solutions[a][n][s] = solution
            times[a][n][s] = time
        log(completed, n, s)

print(" " * 100, end="\r")
print("Done")
```

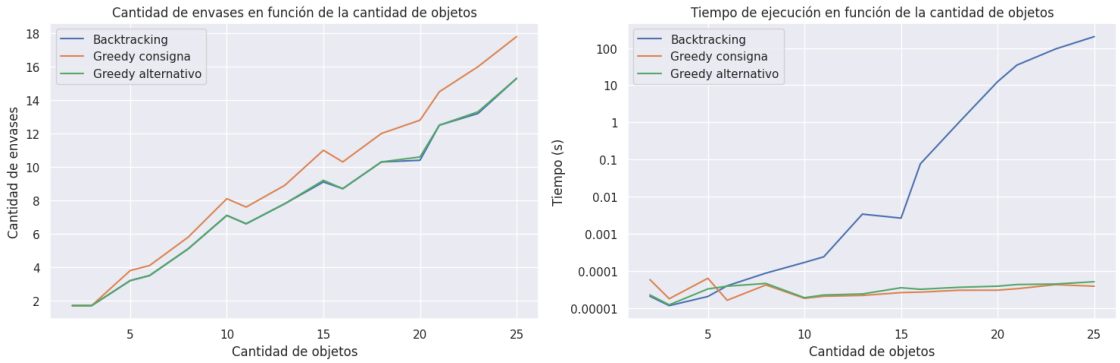
```
Done
```

[31]:
```
import json
import os

def save(path: str, data):
    data = [{"cant_objetos": int(i), "resultados": {a: [int(j) for j in␣
↪data[a][i]] for a in data}} for i in x]
    with open(path, "w") as f:
        f.write(json.dumps(data, indent=2))

if not os.path.exists("pruebas"):
    os.makedirs("pruebas")
```

```
save("pruebas/soluciones.json", solutions)
save("pruebas/tiempos.json", times)
```

Gráficos

[17]:
```
from matplotlib import pyplot as plt
from matplotlib import ticker
import seaborn as sns
sns.set()

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
fig.set_tight_layout(True)

PlotData = dict[Algorithm, dict[int, float]]
def show(ax: plt.Axes, data: PlotData, y_label: str, title: str,  log: bool =␣
↪False):
    for a in data:
        ax.plot(data[a].keys(), data[a].values(), label=str(a))
    ax.set_xlabel("Cantidad de objetos")
    ax.set_ylabel(y_label)
    ax.set_title(title)
    if log:
        ax.set_yscale("log")
        ax.yaxis.set_major_formatter(ticker.FuncFormatter(lambda y,pos: ('{{:.{:
↪1d}f}}'.format(int(np.maximum(-np.log10(y),0))))).format(y)))
    ax.legend()

def avg(data: dict[int, list[int | float]]) -> dict[int, float]:
    return {x: np.mean(measures) for x, measures in data.items()}

show(ax1, {a: avg(s) for a, s in solutions.items()} , "Cantidad de envases",␣
↪"Cantidad de envases en función de la cantidad de objetos")
show(ax2, {a: avg(s) for a, s in times.items()}, "Tiempo (s)", "Tiempo de␣
↪ejecución en función de la cantidad de objetos", log=True)
```



Primer gráfico

El gráfico de la izquierda muestra la solución obtenida promedio para las pruebas, en función de la cantidad de objetos. Podemos ver que el algoritmo Greedy alternativo propuesto, en la mayoría de los casos, obtiene la solución exacta del problema que encontramos con el algoritmo de backtracking. Esto no es así con el algoritmo Greedy de la consigna, que pareciera desviarse cada vez más de la solución exacta a medida que aumenta la cantidad de objetos.

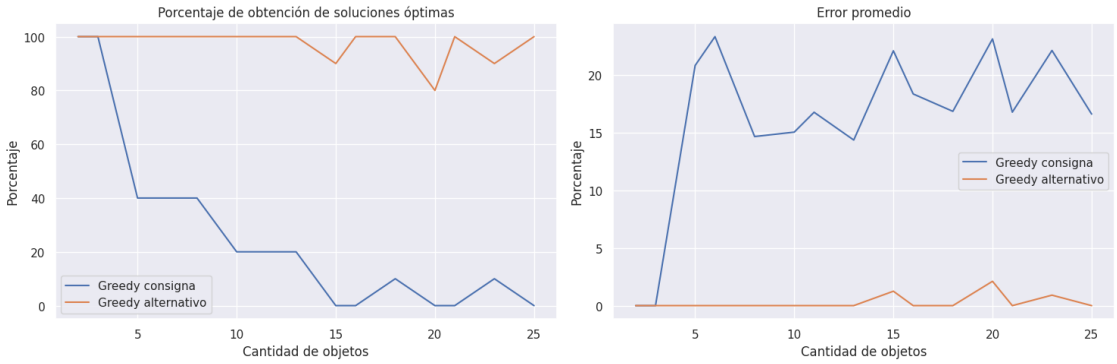
Segundo gráfico

El gráfico de la derecha muestra el tiempo empleado en segundos (escala logarítimica), en función de la cantidad de objetos. Claramente el algoritmo de backtracking toma muchísimo más tiempo que los otros dos, debido a su complejidad exponencial $O(B_n)$. Los algoritmos Greedy tienen un rendimiento muy superior, casi instantáneo para esta cantidad de elementos. El algoritmo de la consigna parece ser levemente más rápido que el alternativo, lo cual es acorde a sus complejidades computacionales ($O(n)$ vs. $O(nlog(n))$). Aun así, es una entrada muy pequeña elementos como para destacar una diferencia notable.

[8]:
```
# Algoritmo -> n objetos -> estadística
right: PlotData = {}
error: PlotData = {}

for a in (Algorithm.GREEDY, Algorithm.GREEDY_ALT):
    r = {n: [100 if s==s_exact else 0 for s, s_exact in zip(solutions[a][n],␣
 ↪solutions[Algorithm.EXACT][n])] for n in x}
    e = {n: [100*(s-s_exact)/s_exact for s, s_exact in zip(solutions[a][n],␣
 ↪solutions[Algorithm.EXACT][n])] for n in x}
    right[a] = avg(r)
    error[a] = avg(e)

    print(f"{a}:")
    print(f"\tSolución óptima el {sum(right[a].values()) / len(x) :.2f}% de las␣
 ↪veces")
    print(f"\tError máximo del {max(x for y in e.values() for x in y):.2f}%")
    print(f"\tError promedio del {sum(error[a].values()) / len(x) :.2f}%\n")

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))
fig.set_tight_layout(True)
show(ax1, right, "Porcentaje", "Porcentaje de obtención de soluciones óptimas")
show(ax2, error, "Porcentaje", "Error promedio")
```

```
Greedy consigna:
        Solución óptima el 26.67% de las veces
        Error máximo del 50.00%
        Error promedio del 16.07%

Greedy alternativo:
        Solución óptima el 97.33% de las veces
        Error máximo del 12.50%
```

```
Error promedio del 0.28%
```



En estas estadísticas y gráficos comparamos exclusivamente los algoritmos Greedy. Podemos ver que para las pruebas realizadas, el algoritmo Greedy de la consigna solo obtuvo la solución óptima alrededor de un cuarto de las veces, a contraste del alternativo que las obtuvo casi siempre.

El error máximo en el algoritmo de la consigna fue del 50%. Era esperable que sea menor a 100% debido a la prueba de la 2-aproximación. El error máximo del alternativo fue mucho menor, 12.5%. Los errores promedio también siguen esta tendencia: 16.07% vs 0.28%.

Primer gráfico

El gráfico de la izquierda muestra el procentaje de soluciones óptimas obtenidas para las muestras con cada cantidad de objetos. Otra vez podemos ver lo mismo: el algoritmo alternativo suele tener 100% y no bajó de 80%, el de la consigna hubo casos que incluso no obtuvo ninguna solución óptima y parece ir empeorando a medida que aumenta la cantidad de objetos.

Segundo gráfico

El gráfico de la derecha muestra el porcentaje de error promedio para las muestras con cada cantidad de objetos. Muy similar al anterior, el alternativo cerca del 0% y el de la consigna va aumentando.

A modo de conclusión, tiene sentido que el algoritmo de la consigna empeore considerablemente a medida que aumentan los objetos: una vez que cierra un envase, ya no lo utiliza más, aunque queden muchísimos objetos que aún puedan entrar en ese envase.