Informe Técnico Completo: Problema de Selección Óptima de Talento

Diseño y Análisis de Algoritmos

6 de enero de 2026

Autores

Javier A. González Díaz
github.com/Javi111003

José M. Leyva de la Cruz
github.com/josemiley02

Kevin Márquez Vega
github.com/kmvega-05

Resumen

Este informe presenta un análisis completo del problema de selección óptima de talento en la plataforma TalentBridge Connect. Se estudian tres enfoques algorítmicos: Backtracking con Poda, Programación Dinámica con Bitmask, y Algoritmo Greedy. Se demuestra que la versión de decisión del problema es NP-Completa (y por tanto la versión de optimización es NP-Hard) y se analizan las complejidades, garantías de optimalidad y aplicabilidad práctica de cada enfoque.

Índice

1. Introducción General

El problema de Selección Óptima de Talento se presenta en el contexto de TalentBridge Connect, una plataforma que conecta clientes con freelancers especializados. El desafío consiste en encontrar el conjunto de empleados de costo mínimo que cubra todos los requisitos de habilidades especificados por un cliente, asegurando que cada habilidad requerida sea cubierta por al menos un empleado con el nivel de dominio mínimo exigido.

Este problema, modelado como una variante del **Weighted Set Cover Problem**, es NP-Hard en su forma de optimización; su versión de decisión es NP-Completa, lo que justifica el estudio de múltiples enfoques algorítmicos con diferentes trade-offs entre optimalidad y eficiencia computacional.

En este informe se presentan tres soluciones:

- **Backtracking con Poda**: Garantiza optimalidad mediante exploración exhaustiva con podas inteligentes
- **Programación Dinámica**: Solución exacta eficiente cuando el número de requerimientos es moderado
- **Algoritmo Greedy**: Aproximación rápida con garantías teóricas de calidad

2. Modelación Matemática del Problema

2.1. Definición Formal

Definición 2.1 (Problema de Cobertura de Habilidades Ponderado)**.** Sea:

- $E = \{e_1, e_2, \ldots, e_n\}$ el conjunto de empleados disponibles
- $S = \{s_1, s_2, \ldots, s_k\}$ el conjunto de habilidades disponibles en el sistema
- Para cada empleado $e_i \in E$:
  - $c_i \in \mathbb{R}_{\geq 0}$ su costo por hora (salario)
  - $L_i(s) \in \{0\} \cup [1, 10]$ el nivel de dominio en la habilidad s, donde 0 indica que no posee la habilidad
- $R = \{(s_j, l_j) : s_j \in S, l_j \in [1, 10]\}$ el conjunto de requerimientos del cliente, donde (s_j, l_j) significa que se requiere habilidad s_j con nivel mínimo l_j

El problema (de optimización) consiste en encontrar un subconjunto $X \subseteq E$ tal que:

$$\text{mín} \sum_{e_i \in X} c_i \tag{1}$$

sujeto a la restricción de cobertura:

$$\forall (s_j, l_j) \in R, \exists e_i \in X : L_i(s_j) \geq l_j \tag{2}$$

Es decir, todo requerimiento debe ser cubierto por al menos un empleado seleccionado con el nivel de dominio suficiente.

Definición 2.2 (Versión de decisión)**.** Dada una cota de presupuesto $B \in \mathbb{R}_{\geq 0}$, la versión de decisión pregunta si existe $X \subseteq E$ tal que:

$$\sum_{e_i \in X} c_i \leq B \quad \text{y} \quad \forall (s_j, l_j) \in R,\ \exists e_i \in X :\ L_i(s_j) \geq l_j.$$

2.2. Relación con Problemas Clásicos de Complejidad

Este problema es una **variante ponderada del Weighted Set Cover Problem** (WSC), uno de los problemas NP-Hard más estudiados en ciencia de la computación.

2.2.1. Mapeo al Problema de Cobertura de Conjuntos

Podemos transformar formalmente nuestro problema en una instancia de WSC:

- El universo U está compuesto por todos los pares (habilidad, nivel): $U = R$
- Cada empleado e_i representa un conjunto $A_i = \{(s, l) \in R : L_i(s) \geq l\}$, es decir, el conjunto de requisitos que puede cubrir
- El peso de cada conjunto es $w(A_i) = c_i$ (costo del empleado)
- El objetivo es encontrar $X \subseteq E$ tal que $\bigcup_{e_i \in X} A_i = U$ minimizando $\sum_{e_i \in X} c_i$

Esta transformación es una reducción polinomial que preserva soluciones óptimas.

2.3. Demostración de NP-Completitud

Teorema 2.1 (NP-Completitud (versión de decisión) y NP-Hardness (optimización))**.** La versión de decisión del problema de selección óptima de talento es NP-Completa. En consecuencia, la versión de optimización es NP-Hard.

Demostración. La demostración procede por reducción polinomial desde **Set Cover Ponderado (WSC)** en su versión de decisión.

Dada una instancia de WSC: $(U, \mathcal{A}, w, B)$ donde:

- $U = \{u_1, u_2, \ldots, u_m\}$ es el universo
- $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ donde cada $A_i \subseteq U$
- $w : \mathcal{A} \to \mathbb{R}_{\geq 0}$ es la función de peso
- B es la cota de presupuesto

construimos una instancia de nuestro problema en tiempo polinomial:

- Para cada $u_j \in U$, creamos una habilidad abstracta s_j con requerimiento $(s_j, 1)$
- Para cada $A_i \in \mathcal{A}$, creamos un empleado e_i con:
  - Costo $c_i = w(A_i)$
  - Habilidades: $L_i(s_j) = 1$ si $u_j \in A_i$, y $L_i(s_j) = 0$ en caso contrario

- El cliente requiere todas las habilidades: $R = \{(s_1, 1), (s_2, 1), \ldots, (s_m, 1)\}$ y el presupuesto es el mismo B.

Existe una cobertura de costo $\leq B$ en WSC si y solo si existe una selección de empleados de costo $\leq B$ que cubra todos los requerimientos.

Finalmente, el problema está en NP porque, dada una solución candidata X, puede verificarse en tiempo polinomial si cubre todos los requerimientos y si el costo total es $\leq B$. □

Corolario 2.1. Bajo la conjetura $P \neq NP$, no existe un algoritmo de tiempo polinomial que resuelva nuestro problema de optimización de forma óptima. Los mejores algoritmos exactos conocidos tienen complejidad exponencial en el peor caso.

2.4. Estructuras de Datos

Definición 2.3 (Habilidad)**.** Una **habilidad** (Skill) es una capacidad profesional discreta. Los niveles de dominio se representan en una escala de 1 a 10.

Definición 2.4 (Empleado)**.** Un **empleado** (Employee) representa un freelancer con atributos: id, nombre, salario por hora, y mapeo de habilidades a niveles de dominio.

Definición 2.5 (Cliente)**.** Un **cliente** (Client) representa una solicitud con requisitos: mapeo de habilidades requeridas a niveles mínimos.

Definición 2.6 (Solución)**.** Una **solución** (Solution) contiene: conjunto de empleados seleccionados, costo total, e indicador de validez.

Parte I
Enfoque 1: Backtracking con Poda

3. Descripción del Algoritmo de Backtracking

El algoritmo de **Backtracking con Poda (Branch and Bound)** explora el espacio de soluciones de forma sistemática, eliminando ramas que demostablemente no pueden conducir a una solución mejor que la actual.

3.1. Pseudocódigo

Algorithm 1 Backtracking con Poda para Selección de Talento

```
1: function BACKTRACK(pos, selected, current_cost)
2:     Poda por Costo: if current_cost ≥ best_solution.cost then return
3:     Verificación de Solución: if is_complete_cover(selected) then
4:         best_solution ← Solution(selected, current_cost)
5:         return
6:     end if
7:     Caso Base: if pos ≥ |employees| then return
8:     end if
9:     Poda por Factibilidad: uncovered ← get_uncovered_requirements(selected)
10:    if ¬can_potentially_cover(pos, uncovered) then return
11:    end if
12:    employee ← employees_list[pos]
13:    Rama 1 - Incluir: if can_cover_any_requirement(employee, uncovered) then
14:        selected.add(employee)
15:        BACKTRACK(pos+1, selected, current_cost + employee.salary)
16:        selected.remove(employee)
17:    end if
18:    Rama 2 - Excluir:
19:        BACKTRACK(pos+1, selected, current_cost)
20: end function
```

3.2. Estrategias de Poda

3.2.1. Poda por Costo (Cost Pruning)

Definición 3.1 (Poda por Costo)**.** Si el costo acumulado current_cost es mayor o igual al mejor costo encontrado (best_cost), la rama se elimina:

$$\text{si current\_cost} \geq \text{best\_cost entonces podar.}$$

Lema 3.1 (Seguridad de la Poda por Costo)**.** Si $c_i \geq 0$ para todo empleado, entonces la poda por costo no elimina ninguna solución mejor que la mejor encontrada hasta el momento.

Demostración. En cualquier rama, agregar empleados solo puede aumentar (o mantener) el costo acumulado. Si current_cost $\geq$ best_cost, ninguna extensión de esa rama puede producir una solución con costo menor que best_cost. □

3.2.2. Poda por Factibilidad (Feasibility Pruning)

Definición 3.2 (Poda por Factibilidad)**.** Si los empleados restantes no pueden cubrir los requerimientos pendientes, la rama se elimina.

Lema 3.2 (Seguridad de la Poda por Factibilidad)**.** Si can_potentially_cover(pos, uncovered) retorna `false` solo cuando *ningún* subconjunto de empleados desde `pos` puede cubrir `uncovered`, entonces la poda por factibilidad no elimina soluciones válidas.

Demostración. Bajo la condición del enunciado, si la función devuelve `false` entonces no existe continuación posible que complete la cobertura; por tanto, podar es correcto. □

3.3. Correctitud del Algoritmo de Backtracking

Teorema 3.1 (Correctitud del Backtracking)**.** Si el algoritmo termina, retorna una solución óptima (si existe) o declara que no existe solución válida.

Demostración. Por construcción, el algoritmo explora recursivamente las decisiones incluir/excluir para cada empleado (completitud). Solo actualiza la mejor solución cuando la cobertura es completa (validez). Las podas son seguras por los lemas anteriores, por lo que no descartan soluciones mejores. Así, la mejor solución registrada al finalizar es óptima. □

3.4. Análisis de Complejidad de Backtracking

Teorema 3.2 (Complejidad Temporal del Peor Caso)**.** La complejidad temporal en el peor caso es $O(2^n \cdot \text{poly}(n, m))$, donde $n = |E|$ y $m = |R|$.

Demostración. El árbol de decisión tiene hasta $2^{n+1} - 1$ nodos. En cada nodo se realizan operaciones polinomiales (verificación de cobertura/actualización de requerimientos y podas). Por tanto, el tiempo es $O(2^n \cdot \text{poly}(n, m))$. □

Proposición 3.1 (Efectividad de la Poda)**.** En instancias prácticas, el número de nodos visitados se reduce a $O(c^n)$ donde $c < 2$.

Teorema 3.3 (Complejidad Espacial)**.** La complejidad espacial es $O(n + m)$.

Parte II
Enfoque 2: Programación Dinámica

4. Descripción del Enfoque de Programación Dinámica

Aunque el problema es NP-Hard en general, cuando el número de requerimientos (m) es moderado, el problema exhibe propiedades que permiten una solución exacta eficiente mediante **Programación Dinámica con Bitmask**.

4.1. Representación mediante Máscaras de Bits

Definición 4.1 (Máscara de Cobertura)**.** Para un subconjunto $T \subseteq R$ de requerimientos, su **máscara de cobertura** es:

$$\text{mask}(T) = \sum_{r_j \in T} 2^j$$

El bit j-ésimo es 1 si y solo si $r_j \in T$.

Definición 4.2 (Máscara de un Empleado)**.** Para un empleado e_i, su **máscara de habilidades** M_i representa los requerimientos que puede satisfacer:

$$M_i = \sum_{\{j : L_i(s_j) \geq l_j\}} 2^j$$

Definición 4.3 (Estado de la tabla DP)**.** $DP[S]$ denota el costo mínimo para cubrir **al menos** todos los requerimientos cuyos bits están en 1 en la máscara S, usando un subconjunto de los empleados ya procesados por el algoritmo (patrón 0/1).

Observación 4.1 (Motivación para DP)**.** Mientras Backtracking tiene complejidad exponencial en n, DP logra $O(n \cdot 2^m)$, exponencial solo en m. En aplicaciones reales, típicamente $m \ll n$.

4.2. Principios de Programación Dinámica

Teorema 4.1 (Subestructura Óptima)**.** Sea $\text{OPT}(S, i)$ el costo mínimo para cubrir los requerimientos en la máscara S usando empleados $\{e_1, \ldots, e_i\}$. Entonces:

$$\text{OPT}(S, i) = \text{mín}\{\text{OPT}(S, i-1), \text{OPT}(S \wedge \neg M_i, i-1) + c_i\} \tag{3}$$

Demostración. Una solución óptima X^* para cubrir S con $\{e_1, \ldots, e_i\}$ considera dos casos:

Caso 1: Si $e_i \notin X^*$, entonces el costo es $\text{OPT}(S, i-1)$.

Caso 2: Si $e_i \in X^*$, entonces $X^* \setminus \{e_i\}$ debe cubrir $S \wedge \neg M_i$ con costo $\text{OPT}(S \wedge \neg M_i, i-1) + c_i$. □

Proposición 4.1 (Superposición de Subproblemas)**.** El número de subproblemas distintos está acotado por 2^m (número de subconjuntos de R), no por 2^n.

4.3. Algoritmo de DP

Algorithm 2 Preprocesamiento de Máscaras

```
1: function ComputeMasks(employees, requirements)
2:     masks ← empty dictionary
3:     for each e ∈ employees do
4:         M_e ← 0
5:         for j ← 0 to |requirements| − 1 do
6:             (s_j, l_j) ← requirements[j]
7:             if e.skill_level(s_j) ≥ l_j then
8:                 M_e ← M_e | (1 ≪ j)
9:             end if
10:        end for
11:        if M_e ≠ 0 then
12:            masks[e] ← M_e
13:        end if
14:    end for
15:    return masks
16: end function
```

Algorithm 3 Programación Dinámica con Bitmask

```
1: function SolveDP(employees, requirements)
2:     m ← |requirements|
3:     FULL_MASK ← (1 ≪ m) − 1
4:     masks ← ComputeMasks(employees, requirements)
5:     DP ← Array[0 ... FULL_MASK] initialized to ∞
6:     parent ← Array[0 ... FULL_MASK] initialized to None
7:     DP[0] ← 0
8:     for each (e, M_e) ∈ masks do
9:         c_e ← e.cost
10:        for S ← FULL_MASK downto 0 do
11:            if DP[S] < ∞ then
12:                next_S ← S | M_e
13:                if DP[S] + c_e < DP[next_S] then
14:                    DP[next_S] ← DP[S] + c_e
15:                    parent[next_S] ← (e, S)
16:                end if
17:            end if
18:        end for
19:    end for
20:    if DP[FULL_MASK] = ∞ then
21:        return (None, ∞)
22:    end if
23:    selected ← ∅
24:    S ← FULL_MASK
25:    while S ≠ 0 do
26:        (e, prev_S) ← parent[S]
27:        selected.add(e)
28:        S ← prev_S
29:    end while
30:    return (selected, DP[FULL_MASK])
31: end function
```

4.4. Correctitud del Algoritmo de DP

Teorema 4.2 (Correctitud del Algoritmo DP)**.** El algoritmo retorna siempre el costo mínimo posible para cubrir todos los requerimientos, si existe una solución factible.

Demostración. Por inducción sobre el número de empleados procesados.

Invariante: Después de procesar i empleados, $DP[S]$ contiene el costo mínimo para cubrir **al menos** los requerimientos en S usando un subconjunto de $\{e_1, \ldots, e_i\}$.

Caso Base: $DP[0] = 0$, $DP[S] = \infty$ para $S \neq 0$.

Paso Inductivo: Al procesar e_i, la iteración inversa implementa la recurrencia 0/1:

$$\mathrm{OPT}(S, i) = \min\{\mathrm{OPT}(S, i-1), \mathrm{OPT}(S \wedge \neg M_i, i-1) + c_i\}.$$

Por tanto, el invariante se preserva y al final $DP[\mathrm{FULL\_MASK}]$ es óptimo. □

4.5. Análisis de Complejidad de DP

Teorema 4.3 (Complejidad Temporal)**.** La complejidad temporal es $\Theta(n \cdot 2^m)$, donde n es el número de empleados y m es el número de requerimientos.

Demostración. **Preprocesamiento**: $O(n \cdot m)$.

Bucle Principal: n iteraciones externas, 2^m estados internos, $O(1)$ por transición: $O(n \cdot 2^m)$.

Reconstrucción: $O(n)$.

Por tanto, $T(n, m) = O(n \cdot 2^m)$. □

Teorema 4.4 (Complejidad Espacial)**.** La complejidad espacial es $\Theta(2^m)$.

4.6. Fixed-Parameter Tractability (FPT)

Teorema 4.5 (FPT con Parámetro m)**.** El problema es FPT cuando se parametriza por el número de requerimientos m.

Demostración. La complejidad $T(n, m) = 2^m \cdot n$ cumple la definición de FPT con $f(k) = 2^k$ y $|I| = n$. □

Parte III
Enfoque 3: Algoritmo Greedy

5. Descripción del Algoritmo Greedy

El algoritmo **Greedy (Codicioso)** es un algoritmo de **aproximación** que intercambia optimalidad por eficiencia computacional.

5.1. Estrategia General

1. **Inicializar**: Comenzar con conjunto vacío de empleados
2. **Iterar**: Mientras existan requerimientos no cubiertos:
   - *a*) Calcular eficiencia de cada empleado disponible
   - *b*) Seleccionar empleado con máxima eficiencia
   - *c*) Actualizar requerimientos pendientes
3. **Terminar**: Cuando no hay más requerimientos o empleados útiles

5.2. Función de Eficiencia

Definición 5.1 (Eficiencia de un Empleado)**.** Para un empleado e_i y requerimientos no cubiertos U:

$$\text{eficiencia}(e_i, U) = \frac{|\{r \in U : e_i \text{ puede cubrir } r\}|}{c_i}$$

Esta métrica favorece empleados que cubren muchos requerimientos a bajo costo.

5.3. Pseudocódigo

Algorithm 4 Algoritmo Greedy para Selección de Talento

```
function GREEDY(employees, requirements)
    selected ← ∅
    available ← employees
    uncovered ← requirements
    total_cost ← 0
    while uncovered ≠ ∅ and available ≠ ∅ do
        best_employee ← nil
        best_efficiency ← 0
        for each e ∈ available do
            coverage ← |{r ∈ uncovered : e cubre r}|
            if coverage > 0 then
                eff ← coverage / c_e                    ▷ c_e es el costo de e
                if eff > best_efficiency then
                    best_employee ← e
                    best_efficiency ← eff
                end if
            end if
        end for
        if best_employee = nil then
            break
        end if
        selected.add(best_employee)
        available.remove(best_employee)
        total_cost ← total_cost + c_best_employee
            ▷ Eliminar de uncovered los requerimientos cubiertos por best_employee
        for each r ∈ uncovered do
            if best_employee cubre r then
                uncovered.remove(r)
            end if
        end for
    end while
    is_valid ← (uncovered = ∅)
    return Solution(selected, total_cost, is_valid)
end function
```

5.4. Garantía de Aproximación

Teorema 5.1 (Garantía de Aproximación del Greedy)**.** Sea C^* el costo óptimo y C_G el costo greedy. Si la solución greedy es válida:

$$C_G \leq C^* \cdot H(m)$$

donde $H(m) = 1 + \frac{1}{2} + \cdots + \frac{1}{m}$ es el m-ésimo número armónico.

Demostración. Este es el resultado clásico para *Set Cover Ponderado* usando selección codiciosa por ganancia marginal/costo. Una demostración completa puede encontrarse en textos estándar de aproximación, por ejemplo [2, 3]. □

Corolario 5.1 (Implicaciones Prácticas)**.** Para problemas con:

- $m = 5$ requerimientos: $H(5) \approx 2{,}28$
- $m = 10$ requerimientos: $H(10) \approx 2{,}93$
- $m = 20$ requerimientos: $H(20) \approx 3{,}60$

El número armónico crece logarítmicamente.

5.5. Correctitud del Algoritmo Greedy

Teorema 5.2 (Validez de la Solución Greedy)**.** Si el algoritmo retorna una solución válida, entonces cubre todos los requerimientos del cliente.

Teorema 5.3 (Completitud del Algoritmo)**.** El algoritmo greedy siempre termina en a lo sumo n iteraciones.

5.6. Análisis de Complejidad del Greedy

Teorema 5.4 (Complejidad Temporal)**.** La complejidad temporal es $O(n^2 \cdot m)$, donde $n = |E|$ y $m = |R|$.

Demostración. El bucle while itera a lo sumo n veces. En cada iteración:

- Búsqueda del mejor empleado: $O(n \cdot m)$
- Actualización de requerimientos: $O(m)$

Total: $T(n, m) = n \times O(n \cdot m) = O(n^2 \cdot m)$. □

Teorema 5.5 (Complejidad Espacial)**.** La complejidad espacial es $O(n + m)$.

6. Comparación de los Tres Enfoques

6.1. Tabla Comparativa de Complejidad

Métrica	**Backtracking**	**DP**	**Greedy**
Mejor caso	$O(\text{poly}(n, m))$	$O(n \cdot 2^m)$	$O(n \cdot m)$
Caso promedio	$O(c^n \cdot \text{poly}(n, m))$	$O(n \cdot 2^m)$	$O(k \cdot nm)$
Peor caso	$O(2^n \cdot \text{poly}(n, m))$	$O(n \cdot 2^m)$	$O(n^2 \cdot m)$
Espacio	$O(n + m)$	$O(2^m)$	$O(n + m)$
Garantía	Óptima	Óptima	$H(m)$-aprox
Ideal para	$n \leq 20$	$m \leq 20$	n o m grandes

6.2. Criterios de Selección

La elección del algoritmo debe basarse en:

- **Backtracking**:
  - Instancias pequeñas ($n \leq 16 - 20$)
  - Cuando se requiere optimalidad garantizada
  - Casos donde la poda es efectiva
- **Programación Dinámica**:
  - Pocos requerimientos ($m \leq 20$)
  - Cuando se requiere optimalidad garantizada
  - Suficiente memoria disponible ($O(2^m)$)
- **Greedy**:
  - Instancias grandes (cualquier n o m)
  - Tiempo de respuesta crítico
  - Aproximación aceptable
  - Recursos computacionales limitados

6.3. Trade-offs Fundamentales

Aspecto	**Backtracking**	**DP**	**Greedy**
Velocidad	Lenta para $n > 20$	Media para $m \leq 20$	Rápida siempre
Optimalidad	Garantizada	Garantizada	Aproximada ($H(m)$)
Escalabilidad	Limitada en n	Limitada en m	Excelente
Memoria	Baja	Alta (2^m)	Baja
Implementación	Moderada	Moderada	Simple

7. Conclusiones Generales

1. La versión de decisión del problema de Selección Óptima de Talento es **NP-Completa**, y su versión de optimización es **NP-Hard**, lo que justifica el estudio de múltiples enfoques con diferentes características.
2. **Backtracking con Poda**:
   - Garantiza optimalidad con complejidad exponencial en n
   - Estrategias de poda reducen significativamente el espacio de búsqueda
   - Viable para instancias pequeñas-medianas ($n \leq 20$)
3. **Programación Dinámica**:
   - Garantiza optimalidad con complejidad $O(n \cdot 2^m)$
   - Es FPT parametrizado por m
   - Viable cuando $m \leq 20$ (típico en aplicaciones reales)
   - Requiere $O(2^m)$ espacio
4. **Algoritmo Greedy**:
   - Complejidad polinomial $O(n^2 \cdot m)$
   - Garantía de aproximación $H(m)$ (logarítmica en m)
   - Excelente escalabilidad para instancias grandes
   - En práctica, brecha con óptimo menor que factor teórico
5. La elección del algoritmo debe considerar:
   - Tamaño de la instancia (n y m)
   - Requisito de optimalidad vs. tiempo
   - Recursos computacionales disponibles
   - Naturaleza del dominio de aplicación
6. Para sistemas de producción, una estrategia híbrida es recomendable:
   - Usar DP cuando $m \leq 20$
   - Usar Backtracking cuando $n \leq 20$ y $m > 20$
   - Usar Greedy como fallback o para respuesta rápida
   - Combinar Greedy con búsqueda local para mejorar calidad

Referencias

[1] Garey, M. R., & Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman.

[2] Vazirani, V. V. (2001). *Approximation Algorithms*. Springer.

[3] Williamson, D. P., & Shmoys, D. B. (2011). *The Design of Approximation Algorithms*. Cambridge University Press.