ANÁLISIS Y DISEÑO DE ALGORITMOS CC371-A
Prof. Jaime Osorio Ubaldo

TÉCNICAS DE DISEÑO DE ALGORITMOS: PROBLEMA DE PLANIFICACIÓN DE TAREAS

—

Por: Guillermo Ronie Salcedo Alvarez

APLICANDO ALGORITMO GREEDY

El problema de la planificación de tareas consiste en que dadas ciertas tareas, se busca asignar recursos limitados a cada una de ellas de la manera más óptima. De este modo, buscamos minimizar el tiempo de las tareas, nuestros costos, bajo la condición de que nuestros recursos, en este caso, trabajadores, solo pueden realizar una tarea, y que una tarea solo puede ser realizada por un trabajador.

Dado el contexto y las condiciones del problema, procederemos a resolverlo por diversas técnicas de diseño de algoritmos: greedy, backtracking y ramificación y poda. Empezaremos por greedy, como aquel que toma decisiones basándose en información inmediata.

```
tareasGreedy(matrix costos)

       Inicializar vector trabDisp a 0
       Definir vector de pares solucion

       Para cada tarea:
              int minCosto = INF
              int mejorTrab = 0

              Para cada trabajador:
                     Si costo[trabajador][tarea] < menorCosto
                     y trabDisp[trabajador] == 0:
                            mejorTrab = trabajador
                            minCosto = costos[trabajador][tarea]

              Insertar el par (mejorTrab, tarea)
              trabDisp[mejorTrab] = 1

       int costoTotal = suma de costos[trabajador][tarea] del vector solucion
       Devuelve solucion, costoTotal

fin-tareasGreedy
```

Resolveremos nuestro problema bajo dicha técnica con el siguiente ejemplo:

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

```
tareasGreedy(costos)

       trabDisp = [0, 0, 0, 0]
       solucion = [(0, 0)]

       trabDisp = [1, 0, 0, 0]
       solucion = [(0, 0), (3, 1)]

       trabDisp = [1, 0, 0, 1]
       solucion = [(0, 0), (3, 1), (1, 2)]

       trabDisp = [1, 1, 0, 1]
       solucion = [(0, 0), (3, 1), (1, 2), (2, 3)]
```

```
trabDisp = [1, 1, 1, 1]
solucion = [(0, 0), (3, 1), (1, 2), (2, 3)]
```

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

APLICANDO ALGORITMO BACKTRACKING

Bajo la técnica de diseño de backtracking, lo que buscamos es encontrar todas los subconjuntos posibles para la solución del problema, y una vez encontrado un resultado no viable, se realiza un retroceso. A continuación el pseudocódigo correspondiente:

```
backtrack(matrix costos, solActual, trabDisp, solucion, mejorCostoTotal)

    Si length(solActual) == length(tareas):
        costoTotal = suma de costos[trabajador][tarea] del vector solActual

        Si costoTotal < mejorCostoTotal:
            mejorCostoTotal = costoTotal
            solucion = solActual
        Devuelve solucion, mejorCostoTotal

    Para cada trabajador:
        Insertar trabajador en solActual
        Remover trabajador de trabDisp

        solucion, mejorCostoTotal = backtrack(costos, solActual, trabDisp,
        solucion, mejorCostoTotal)

        Remover el último elemento de solActual
        Insertar trabajador a trabDisp

    Devuelve solucion, mejorCostoTotal

fin-backtrack

tareasBacktracking(costos)
```

```
        Inicializar el conjunto trabDisp
        Definir solucion, solActual = []
        int mejorCostoTotal = INF

        solucion = backtrack(costos, solActual, trabDisp, solucion, mejorCostoTotal)
        Definir soluciones como pares (trabajador, tarea) en solucion

        Devuelve soluciones, mejorCostoTotal

fin-tareasProb
```

Resolveremos el ejemplo anterior con este nuevo método:

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

```
tareasBacktracking(costos)
        backtrack(costos, [], [0, 1, 2, 3], solucion, INF)
                solActual = [0]
                trabDisp = [1, 2, 3]
                solucion = backtrack(costos, [0], [1, 2, 3], solucion, INF)

        backtrack(costos, [0], [1, 2, 3], solucion, INF)
                solActual = [0, 1]
                trabDisp = [2, 3]
                solucion = backtrack(costos, [0, 1], [2, 3], solucion, INF)

        (...)

        backtrack(costos, [0, 1, 2, 3], [], solucion, INF)
                costoTotal = 73
                mejorCostoTotal = 73
                solucion = solActual
                Devuelve solucion, mejorCostoTotal
```

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

(Se realiza un retroceso y busca si es posible otra combinación)

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

(Así sucesivamente se calculan todas las posibles combinaciones)

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

APLICANDO ALGORITMO BRANCH AND BOUND

Bajo la técnica de diseño de ramificación y poda (*branch and bound*), una variante de backtracking, que utiliza cotas (superiores e inferiores) para poder eliminar (podar) ramas que no nos conduzcan a una solución óptima.

```
tareasRamificacion(matrix costos)

	Definir cotaSup = min(diagonal, diagonalSec)
	Definir cotaInf = max(minFilas, minColumnas)

	Para cada tarea:
		Si cotaInf > arbParcial:
			cotaInf = arbParcial

	Mientras que lista no vacía:
		Elegir rama de menor coste
		Generar ramas del menor
		Podar ramas

		Para cada trabajador:
			Si costos(rama) > costos(mejorAct)
				Se poda rama
			Sino
				Si (no es solucion):
					Pasar a lista de nodos vivos
				Sino
					Se encontró mejor solución
					Podar nodos peores

fin-tareasRamificacion
```

Resolvemos con el ejemplo anterior. Se calculan las soluciones parciales asignando al trabajador a, la primera tarea, y se asignan los menores costos de las tareas. Esto se repite para cada una de las tareas.

- Árboles parciales:
  - **11** + 14 + 13 +22 = 60
  - 11 + 12 + 13 + 22 = 58
  - 18 + 11 + 14 + 22 = 65
  - 40 + 11 + 14 + 13 = 78

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

- Árboles parciales: 68, **59**, 64

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

- Árboles parciales: **64**, 65

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

Se poda porque es mayor que el árbol de solución parcial.

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28

- Árboles parciales: 68, **58**, 66

Trab/Tarea	1	2	3	4
A	11	12	18	40
B	14	15	13	22
C	11	17	19	23
D	17	14	20	28