

# Informe de Tarea 1 - CIT3352

## Algoritmos Exactos y Metaheurísticas

Nombre: Lucas Gonzalez y Diego Pastrian  
Profesor: Víctor Reyes Rodríguez

1 de abril de 2025

### Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Modelado del Problema</b>	<b>2</b>
2.1. Modelo asociado al problema . . . . .	2
2.2. Datos del problema . . . . .	3
2.3. Espacio de búsqueda del problema . . . . .	4
<b>3. Desarrollo e Implementación</b>	<b>4</b>
3.1. Búsqueda Completa (Forward-Checking) . . . . .	4
3.2. Variante con Heurística de Selección de Variable . . . . .	7
<b>4. Resultados y Análisis</b>	<b>8</b>
4.1. Resultados de la Ejecución . . . . .	8
4.2. Análisis Comparativo . . . . .	10
<b>5. Conclusiones</b>	<b>11</b>
<b>6. Referencias</b>	<b>11</b>

# 1. Introducción

En este informe se presenta el desarrollo y resolución de la Tarea 1, en la cual se plantea el problema de instalación de centros de vacunación en la región Brisketiana. Dado que se debe garantizar la cobertura de todas las comunas minimizando el costo de construcción, se ha modelado el problema utilizando variables binarias, función objetivo y restricciones de cobertura. Para resolver el problema se implementaron dos formas:

1. Un algoritmo exacto basado en búsqueda completa (branch & bound).
2. Una variante que incorpora una heurística de selección de variable mediante un índice de utilidad.

## 2. Modelado del Problema

### 2.1. Modelo asociado al problema

El problema se modela de la siguiente manera:

■ **Conjuntos de referencia:**

- $\mathcal{N} = \{1, 2, \dots, n\}$ : Conjunto Comunas de la región Brisketiana.

■ **Variables de decisión:**

$$x_i = \begin{cases} 1, & \text{si se construye un centro en la comuna } i, \\ 0, & \text{en caso contrario.} \end{cases}$$

■ **Parámetros:**

$C_i$  : Costo asociado a la construcción del centro en la comuna  $i$ .

$$T_{ij} = \begin{cases} 1, & \text{si la comuna } i \text{ puede satisfacer la demanda de la comuna } j \\ 0, & \text{en caso contrario.} \end{cases}$$

■ **Función Objetivo:** Minimizar el costo total:

$$\text{mín} \sum_{i=1}^n C_i \cdot x_i.$$

■ **Restricciones:**

$$\sum_{j=1}^n T_{ij} \cdot x_i \geq 1, \forall j \in \mathcal{N} \quad (\text{Cobertura por comuna})$$

$$x_i \in \{0, 1\}, \quad \forall i \in \mathcal{N} \quad (\text{Dominio de las variables})$$

## 2.2. Datos del problema

En esta sección, se describen los parámetros y matrices que definen el modelo, específicamente los costos asociados a la construcción de los centros de vacunación y la cobertura entre las comunas de la región Brisketiana. Para el problema del enunciado, la región está compuesta por  $n = 15$  comunas, y se necesita establecer un modelo para minimizar el costo total de construcción de los centros de vacunación mientras se asegura que todas las comunas estén cubiertas.

- **Comunas y Costo:** El primer parámetro del modelo es el costo  $C_i$  asociado con la construcción de un centro de vacunación en cada comuna  $i$ . Los valores de este parámetro están dados en la siguiente tabla:

Cuadro 1: Tabla de Costos  $C_i$

Comuna $i$	Costo $C_i$	Comuna $i$	Costo $C_i$
1	60	9	80
2	30	10	70
3	60	11	50
4	70	12	90
5	130	13	30
6	60	14	30
7	70	15	100
8	60		

- **Matriz de Cobertura  $T_{ij}$ :** El segundo parámetro del modelo es la matriz de cobertura  $T_{ij}$ , que indica si la comuna  $i$  puede satisfacer la demanda de la comuna  $j$ . A continuación se presentan los datos de este parámetro para el problema del enunciado.

Cuadro 2: Matriz de Cobertura  $T_{ij}$ 

$i/j$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1	1	1	1	0	0	0	0	0	0	0	0	1	0	0
2	1	1	0	1	0	0	0	0	0	0	0	1	0	0	1
3	1	0	1	1	1	1	0	0	0	0	0	0	1	0	0
4	1	1	1	1	1	0	0	0	0	0	0	1	0	0	0
5	0	0	1	1	1	1	1	1	1	0	0	1	0	0	0
6	0	0	1	0	1	1	0	0	1	0	0	0	0	0	0
7	0	0	0	0	1	0	1	1	0	1	1	1	0	1	1
8	0	0	0	0	1	0	1	1	1	1	0	0	0	0	0
9	0	0	0	0	1	1	0	1	1	1	1	0	0	0	0
10	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
11	0	0	0	0	0	0	1	0	1	1	1	0	0	1	0
12	0	1	0	1	1	0	1	0	0	0	0	1	0	0	1
13	1	0	1	0	0	0	0	0	0	0	0	0	1	0	0
14	0	0	0	0	0	0	1	0	0	0	1	0	0	1	1
15	0	1	0	0	0	0	1	0	0	0	0	1	0	1	1

### 2.3. Espacio de búsqueda del problema

Dado que cada una de las 15 comunas puede tener o no un centro de búsqueda, el espacio de búsqueda es de:

$$2^{15} = 32,768$$

posibles asignaciones. Esto quiere decir que existen 32,768 posibles soluciones que el modelo debería evaluar para encontrar la mejor.

## 3. Desarrollo e Implementación

Para resolver el problema se desarrollaron dos técnicas en Python.

### 3.1. Búsqueda Completa (Forward-Checking)

La solución para el problema se utiliza Forward checking, método el cual explora todas las posibles asignaciones para las variables de instalación de los centros de vacunación, descartando anticipadamente aquellas donde no se puede llegar a una asignación válida. Esto se realiza de manera de árbol utilizando recurrencia en código, siempre evaluando el caso de instalar la siguiente variable y no instalarla.

La función *busqueda\_forward\_checking* implementa un algoritmo recursivo con poda para minimizar el costo de construcción de los centros de vacunación. Utiliza **forward checking** para descartar soluciones inviables antes de explorarlas, asegurando que aún se puedan cubrir todas las comunas con los centros restantes. Si se encuentra una solución factible y más barata, se actualiza. Se prueban dos opciones para cada comuna: construir o no un centro, aplicando **backtracking** para explorar diferentes combinaciones de manera eficiente y reducir el tiempo de búsqueda.

```
def busqueda_forward_checking(index, solucion_actual, costo_actual, mejor_sol, mejor_costo, inicio, evolucion,
orden):
    """
    Aplica forward checking en cada nivel:
    - Si no se puede cubrir las comunas restantes con lo no asignado, se poda.
    """
    global nodos_visitados_forward

    if costo_actual >= mejor_costo[0]:
        return

    # Comprobamos forward checking
    if not forward_check(solucion_actual):
        return

    if index == len(comunas):
        if es_factible(solucion_actual) and costo_actual < mejor_costo[0]:
            mejor_costo[0] = costo_actual
            mejor_sol.clear()
            mejor_sol.update(solucion_actual)
            evolucion.append((time.time() - inicio, costo_actual))
            return

    comuna = orden[index]

    # no construir centro
    solucion_actual[comuna] = 0
    nodos_visitados_forward += 1
    busqueda_forward_checking(index + 1, solucion_actual, costo_actual,
                                mejor_sol, mejor_costo, inicio, evolucion, orden)

    # construir centro
    solucion_actual[comuna] = 1
    nuevo_costo = costo_actual + costos[comuna - 1] # Ajuste en el índice
    nodos_visitados_forward += 1
    busqueda_forward_checking(index + 1, solucion_actual, nuevo_costo,
                                mejor_sol, mejor_costo, inicio, evolucion, orden)

    # Backtrack
    solucion_actual.pop(comuna)
```

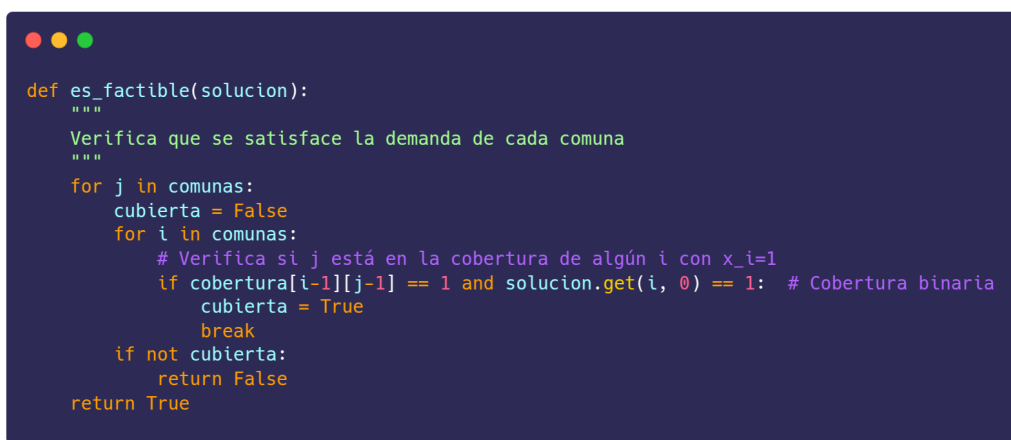
Figura 1: Función *busqueda\_forward\_checking*

La función *forward\_check* verifica si la solución parcial actual aún permite cubrir todas las comunas. Primero, identifica qué comunas no tienen un centro asignado. Luego, revisa si cada comuna ya está cubierta por la asignación actual. Si una comuna no está cubierta, se comprueba si aún existen opciones viables para cubrirla en futuras asignaciones. Si ninguna de las comunas restantes puede cubrirla, se devuelve **False**, indicando que esta solución no es válida y debe descartarse. Esto ayuda a reducir el espacio de búsqueda al evitar soluciones imposibles.

```
def forward_check(solucion):  
    """  
    Observa si cada comuna j podrá ser cubierta a futuro por comuna i  
    """  
    unassigned = [i for i in comunas if i not in solucion]  
    for j in comunas:  
        # ¿j está cubierta por la asignación actual?  
        cubierta_actual = any(  
            cobertura[i-1][j-1] == 1 and solucion.get(i, 0) == 1  
            for i in solucion  
        )  
        if not cubierta_actual:  
            # j no está cubierta: debe haber al menos un 'i' en unassigned que pueda cubrirla  
            if not any(cobertura[i-1][j-1] == 1 for i in unassigned):  
                return False  
    return True
```

Figura 2: Función *forward\_check*

La función *es\_factible* verifica si todas las comunas están cubiertas en la solución actual. Para ello, recorre cada comuna y comprueba si al menos un centro de vacunación cubre su demanda. Si alguna comuna no está cubierta, la función devuelve **False**, indicando que la solución no es válida. En caso contrario, devuelve **True**, confirmando que la asignación actual satisface las restricciones del problema. Esta verificación es fundamental para asegurar que las soluciones generadas sean viables.

A screenshot of a code editor with a dark blue background and light green text. The code defines a function named `es_factible` that takes a `solucion` parameter. It includes a docstring in Spanish: "Verifica que se satisface la demanda de cada comuna". The function iterates over each comuna `j` in the `comunas` list. For each `j`, it sets `cubierta = False` and then iterates over each comuna `i`. Inside this loop, it checks if `cobertura[i-1][j-1] == 1` and `solucion.get(i, 0) == 1`. If both are true, it sets `cubierta = True` and breaks the inner loop. After the inner loop, if `cubierta` is still `False`, it returns `False`. If it completes the loop for all `j`, it returns `True`.

```
def es_factible(solucion):  
    """  
    Verifica que se satisface la demanda de cada comuna  
    """  
    for j in comunas:  
        cubierta = False  
        for i in comunas:  
            # Verifica si j está en la cobertura de algún i con x_i=1  
            if cobertura[i-1][j-1] == 1 and solucion.get(i, 0) == 1: # Cobertura binaria  
                cubierta = True  
                break  
        if not cubierta:  
            return False  
    return True
```

Figura 3: Función *es\_factible*

### 3.2. Variante con Heurística de Selección de Variable

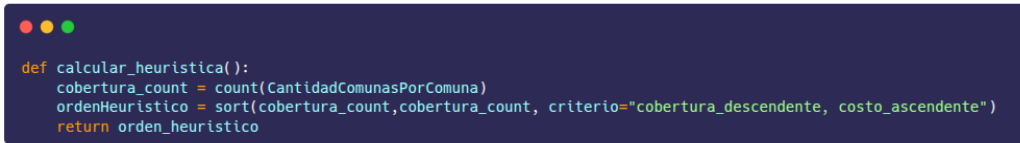
Para mejorar los tiempos de búsqueda se propone una heurística que reordena la selección de las variables según un índice de utilidad definido por:

- Definir primero para las comunas que tenga más vecinos, ya que su cobertura afecta más en la restricción de cobertura.
- Si existe un empate entre la cantidad de comunas vecinas entre una comuna  $i$  y una comuna  $j$ , se prioriza la comuna con el menor costo.

La heurística de priorizar primero la satisfacción de la demanda y luego el costo se enfoca en que las zonas más críticas estén cubiertas desde el principio, evitando recorridos innecesarios y minimizando el riesgo de no cubrir necesidades esenciales. Esto reduce la cantidad de nodos recorridos, ya que

al cubrir las demandas más altas primero, se evita el retroceso o la redistribución de recursos en etapas posteriores. Además, al asignar recursos de manera eficiente desde el inicio, se optimizan los costos, ya que se aprovechan las economías de escala y se evitan asignaciones costosas en áreas de baja prioridad. Así, se obtiene una solución más eficiente tanto en términos de tiempo como de costos. Su implementación como pseudocódigo se observa en la figura 4.

---

A dark-themed code editor window with three colored window control buttons (red, yellow, green) in the top-left corner. It contains a Python function definition for 'calcular\_heuristica()' which calculates a heuristic order for variable selection based on coverage and cost.

```
def calcular_heuristica():  
    cobertura_count = count(CantidadComunasPorComuna)  
    ordenHeuristico = sort(cobertura_count, cobertura_count, criterio="cobertura_descendente, costo_ascendente")  
    return orden_heuristico
```

Figura 4: Pseudocódigo de la heurística de selección de variable.

## 4. Resultados y Análisis

Se muestran los resultados al evaluar bajo este contexto las dos variantes del método Forward Checking para cubrir todas las comunas: una con el orden natural y otra utilizando un orden heurístico basado en el número de vecinos y el costo.

### 4.1. Resultados de la Ejecución

En la figura 5 se pueden observar los resultados en consola de la ejecución del código.



```

o diegoo@diegoo:~/Escritorio/metaheuristicas/Tareal$ python3 tareal.py

=====
Ejecución sin Heurística
=====
- Costo Óptimo: 170
- Tiempo: 10.79 milisegundos
- Centros construidos en comunas: 2, 9, 13, 14
- Nodos visitados: 886
=====

=====
Ejecución con Heurística
=====
El orden de asignación de variables heurístico: [7, 5, 3, 4, 9, 12, 2, 11, 1, 8, 10, 15, 14, 6, 13]
- Costo Óptimo: 170
- Tiempo: 8.91 milisegundos
- Centros construidos en comunas: 9, 2, 14, 13
- Nodos visitados: 660
=====

```

Figura 5: Resultados de la ejecución del código.

Se recopilaron los siguientes valores tras ejecutar ambas versiones del algoritmo: Ante cada ejecución del código el tiempo de ejecución cambia

Método	Costo Óptimo	Tiempo de Ejecución (ms)	Nodos Visitados
Forward Checking (sin heurística)	170	10.79	886
Forward Checking (con heurística)	170	6.51	660

Cuadro 3: Comparación de rendimiento entre las versiones con y sin heurística

debido a las condiciones de la CPU, pero los datos como el Costo óptimo y Nodos visitados siempre se mantienen idénticos.

- **Costo Óptimo:** Representa la solución más barata encontrada.
- **Tiempo de Ejecución:** Mide la eficiencia computacional del algoritmo.
- **Nodos Visitados:** Indica cuántas soluciones parciales se exploraron antes de llegar a la mejor solución.
- **Comunas con Centros:** Lista de comunas donde se instalaron los centros de vacunación.

Adicionalmente, en la Figura 6 se muestra un gráfico comparativo para ambas variantes que representan la mejora del costo al seleccionar el valor de las variables a lo largo del tiempo durante la ejecución del modelo.

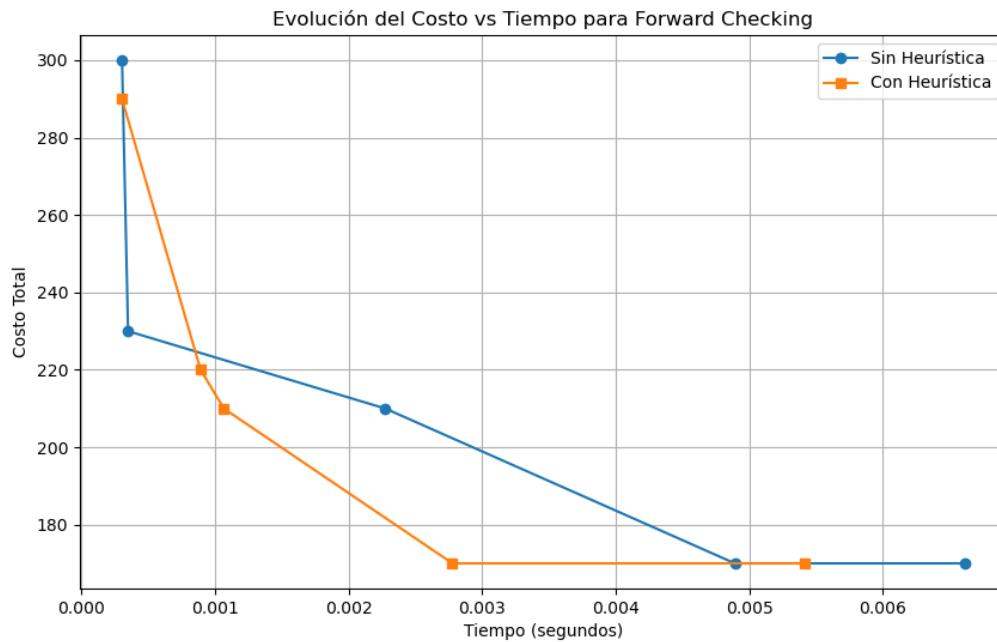


Figura 6: Gráfico comparativo costo/tiempo.

## 4.2. Análisis Comparativo

En la variante sin heurística, el algoritmo recorrió el espacio de soluciones en orden natural, lo que llevó a visitar un mayor número de nodos y a un tiempo de ejecución superior para alcanzar la solución óptima.

Por otro lado, la variante con heurística optimiza la selección de variables, ya que el orden de asignación está establecido bajo el contexto del problema, descartando tempranamente ramas inviables y reduciendo tanto el tiempo de ejecución como la cantidad de nodos explorados. Se logró alcanzar la misma solución óptima en menos tiempo, con una reducción del número de nodos visitados.

En la gráfica se observa que:

- El modelo con la heurística logra encontrar las asignaciones de variables óptimas antes que el modelo sin heurística.
- Con heurística, el modelo termina antes de recorrer el árbol, ya que visita menos nodos que el modelo sin heurística.
- La heurística permite **descartar soluciones inviables rápidamente**, mejorando la eficiencia sin sacrificar la calidad de la solución.

En resumen, ambos métodos encontraron soluciones factibles que cubren todas las comunas, pero la incorporación de la heurística podría mejorar la eficiencia, evidenciado en la evolución más abrupta del costo en la gráfica. Estos resultados demuestran que aplicar un enfoque informado para seleccionar variables mejora significativamente el rendimiento del algoritmo, reduciendo la complejidad computacional y acelerando la convergencia a la solución óptima.

## 5. Conclusiones

En este trabajo se modeló el problema de la instalación de centros de vacunación en la región Brisketiana mediante variables binarias, definiendo una función objetivo que minimiza el costo total y estableciendo restricciones de cobertura para asegurar que cada comuna esté atendida.

Se implementó una técnica de búsqueda Forward-Checking y se desarrolló una variante que incorpora una heurística de selección de variables. Esta técnica permite descartar tempranamente ramas inviables y reducir el número de nodos explorados, mejorando notablemente el tiempo de ejecución, al igual que una heurística de selección de variable elegida correctamente en base al contexto de un problema podría reducir significativamente la cantidad de nodos visitados en un problema de optimización.

En este contexto, la comparación de ambas estrategias evidenció que la incorporación de heurísticas acelera la convergencia a la solución óptima sin sacrificar la calidad del resultado. Este enfoque, que combina algoritmos exactos y heurísticos, es escalable y aplicable a problemas de optimización de mayor envergadura, demostrando su efectividad en la resolución de desafíos complejos.

## 6. Referencias

- 1 Papadimitriou, C. H., & Steiglitz, K. (1998). Combinatorial Optimization: Algorithms and Complexity. Dover Publications..
- 2 Blum, C., & Roli, A. (2003). Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. ACM Computing Surveys, 35(3), 268-308.
- 3 Korte, B., & Vygen, J. (2018). Combinatorial Optimization: Theory and Algorithms. Springer.