UNIVERSIDAD SIMÓN BOLÍVAR
DEPARTAMENTO DE COMPUTACIÓN Y TECNOLOGÍA DE LA INFORMACIÓN
CI5652 - DISEÑO DE ALGORITMOS II
PERÍODO ENERO - MARZO 2026

Informe
Primer Corte del Proyecto

Rafael Valera, 16-11202 - Franco Murillo, 16-10782 - Laura León, 17-10307
Baudilio Velásquez, 18-10665 - Leonardo Dolande, 19-10181
Anya Marcano, 19-10336

RESUMEN

En este informe abordamos el problema de *Minimum Vertex Cover* (MVC), un clásico NP-completo, y presentamos un conjunto de métodos exactos y aproximados para resolverlo. Implementamos una búsqueda exacta por ramificación y una variante *branch-and-bound*, además de una heurística especializada (Isolation Algorithm) y técnicas metaheurísticas basadas en búsqueda local, ILS y GLS. Evaluamos el desempeño sobre instancias DIMACS (usando grafos complementarios) y reportamos costo, gap respecto al óptimo conocido y tiempo promedio. Los resultados muestran que la heurística es extremadamente rápida pero menos precisa en instancias densas, la búsqueda local estabiliza la calidad con bajo costo temporal, e ILS ofrece el mejor equilibrio entre calidad y tiempo bajo límites fijos; GLS actúa como contraste pero no supera a ILS en este conjunto de parámetros.

INTRODUCCIÓN

La motivación principal de este trabajo es estudiar un problema NP-completo clásico que permita contrastar con claridad enfoques exactos y aproximados en grafos. Elegimos el *Minimum Vertex Cover* (MVC) porque es un caso emblemático en optimización combinatoria, aparece en

múltiples aplicaciones reales y, además, cuenta con instancias estándar (DIMACS) y óptimos conocidos que facilitan la comparación objetiva de resultados. Esta combinación lo convierte en un excelente banco de pruebas para analizar el equilibrio entre calidad de solución y costo computacional.

Con base en ello, desarrollamos dos métodos exactos (*branching* y *branch-and-bound*) para establecer una línea base de optimalidad, y varios métodos heurísticos y metaheurísticos para escalar a instancias más grandes. En particular, implementamos el Isolation Algorithm, una búsqueda local con intercambios en dos etapas, y dos esquemas de alto nivel (ILS y GLS) que reutilizan la búsqueda local para intensificar o diversificar la exploración. El objetivo principal es comparar estas estrategias en un entorno controlado usando instancias DIMACS y métricas de costo, gap y tiempo de ejecución, evaluando además la sensibilidad a parámetros como el límite de tiempo y el número de corridas.

DEFINICIÓN DEL PROBLEMA: MINIMUM VERTEX COVER (MVC)

El problema de *Minimum Vertex Cover* (MVC) es uno de los pilares fundamentales en la teoría de grafos y la optimización combinatoria, cuya relevancia histórica y computacional se consolidó en 1972, cuando Richard Karp lo incluyó en su célebre lista de los 21 problemas NP-completos, demostrando su complejidad mediante reducciones polinómicas [1]. Desde entonces, el MVC ha servido como un modelo estándar para el estudio de la complejidad computacional y el desarrollo de nuevas técnicas algorítmicas, siendo documentado exhaustivamente en textos de referencia [2].

Más allá de su interés teórico, el MVC es un problema de decisión y optimización que captura la esencia de la 'cobertura eficiente' y en sí, representa el desafío de seleccionar la mínima cantidad de puntos estratégicos para monitorear, controlar o conectar una red completa de interacciones.

Es más, debido a que el número de posibles combinaciones de vértices crece de forma exponencial con el tamaño del grafo ($2^{|V|}$), el MVC se ha convertido en el *benchmark* para evaluar la eficiencia de algoritmos heurísticos y metaheurísticos en entornos de cómputo de alto rendimiento.

Formalización Matemática

Dado un grafo no dirigido $G = (V, E)$, un *vertex cover* (cobertura de vértices) se define como un subconjunto de vértices $C \subseteq V$ tal que toda arista del grafo es incidente a al menos un vértice del conjunto C.

Formalmente, C es un *vertex cover* si cumple la condición:

$$\forall \{u, v\} \in E : (u \in C \vee v \in C)$$

Siendo así, **el objetivo del problema MVC consiste en contrar un conjunto C cuya cardinalidad $|C|$ sea mínima.** Intuitivamente, lo que queremos es buscar el número mínimo de nodos necesarios para 'cubrir' o supervisar todas las conexiones (aristas) del sistema.

Propiedades y Complejidad

A pesar de ser un problema *NP-hard* en grafos generales, el MVC presenta propiedades teóricas de gran interés, como por ejemplo:

- **Dualidad con *Independent Set*:** Existe una relación fundamental entre el MVC y el problema del *Maximum Independent Set* (MIS). Un conjunto C es un *vertex cover* si y solo si su complemento $V \setminus C$ es un conjunto independiente en G, es decir, un conjunto de vértices sin aristas entre ellos. Esta propiedad permite abordar el problema desde una perspectiva complementaria [4].

- **Grafos bipartitos:** En esta familia de grafos, el MVC pierde su naturaleza intratable y puede resolverse en tiempo polinómico. Esto es posible gracias al **Teorema de Kőnig**, el cual establece que el tamaño del cover mínimo es igual al tamaño del matching máximo [5, 6], es decir, se puede encontrar una solución óptima utilizando algoritmos de matching como el de Hopcroft-Karp.

- **Aproximabilidad:** Dado que hallar la solución exacta es costoso para grafos de gran escala, se suelen emplear esquemas de aproximación, en otras palabras, algoritmos que garantizan encontrar soluciones cercanas al óptimo en tiempo razonable, pero que no aseguran optimalidad.

  Un resultado clásico es el algoritmo de aproximación 2-óptimo basado en matchings maximales, el cual garantiza encontrar un cover no mayor al doble del óptimo en tiempo lineal [3].

Aplicaciones en la Ciencia y la Ingeniería

La capacidad del MVC para modelar problemas de selección mínima lo hace indispensable en diversos campos científicos:

- **Ciberseguridad y redes:** Se utiliza para la ubicación óptima de Sistemas de Detección de Intrusos (IDS), ya que al modelar una red informática como un grafo, el MVC permite identificar el número mínimo de nodos donde instalar sensores para monitorear todo el tráfico de las comunicaciones (aristas) de la red [7].

- **Bioinformática:** En el análisis de datos genómicos, el MVC ayuda a eliminar redundancias en mapas de secuencias de ADN y también es crucial en la biología de sistemas para identificar 'nodos conductores' en redes de interacción de proteínas, donde se busca el conjunto mínimo de proteínas que interactúan con todas las demás en un complejo biológico [8].

- **Análisis de redes sociales:** El MVC se emplea para identificar influenciadores o nodos clave que pueden supervisar o difundir información a través de todas las conexiones sociales de un grupo, facilitando el estudio de la propagación de procesos o virus [9].

En sí, el problema de Minimum Vertex Cover no solo representa un desafío teórico en la ciencia de la computación, sino que también ofrece soluciones prácticas y eficientes para problemas complejos en múltiples disciplinas, demostrando su relevancia y aplicabilidad en el mundo real y al ser un problema NP-completo, motiva la búsqueda constante de algoritmos innovadores que puedan aproximar soluciones óptimas en tiempos razonables.

MARCO EXPERIMENTAL: BENCHMARK SELECCIONADO

Para evaluar qué tan bien funcionan nuestros algoritmos, no basta con probarlos una sola vez. Necesitamos una forma estándar de comparar resultados y entender cómo se comportan ante diferentes tipos de problemas. Siguiendo ejemplos de trabajos previos en el área [11], hemos diseñado un plan de pruebas basado en dos puntos principales: usar grafos con distintas características y analizar cómo afecta el azar a los resultados.

Instancias de Prueba (DIMACS)

Como base de nuestros experimentos, seleccionamos un grupo de grafos del famoso desafío *DIMACS Second Challenge*, los cuales son el estándar en la comunidad científica para probar algoritmos de cobertura y en particular elegimos estas instancias (ver Cuadro) porque mezclan diferentes tamaños y densidades, esto nos permite ver si el algoritmo es rápido en grafos pequeños pero falla en los más densos o complejos, como los de la familia Hamming. Una ventaja clave es que, para la mayoría de estos grafos, ya conocemos la solución óptima, lo que nos permite medir exactamente qué tan cerca (o lejos) estamos del resultado ideal.

Cuadro 0.1: Descripción de los grafos seleccionados para las pruebas.

Instancia	**Vértices** ($\|V\|$)	**Aristas** ($\|E\|$)	**Óptimo (MVC)**
C125.9	125	6,963	91
p_hat300-1	300	10,933	292
DSJC500.5	500	125,248	487
hamming8-4	256	20,864	240
hamming10-4	1,024	434,176	984
gen200_p0_9_44_b	200	17,978	156
keller4	171	9,435	160
keller5	776	225,990	749
MANN_a27	378	70,551	252
brock200_2	200	9,876	188
brock400_2	400	59,786	371
brock800_2	800	208,166	776

Es importante destacar que estas instancias no fueron usadas directamente en los algoritmos, sino que se uso el Grafo Complementario. Esto es porque estas instantacias provenienen de problemas de 'Maximum Clique'. Y como el MVC y el 'Maximum Clique' son problemas complementarios. El optimo del MVC en el grafo complementario es igual al optimo del 'Maximum Clique' en el grafo original.

Protocolo de Ejecución y Aleatoriedad

Muchos de los algoritmos que usamos tienen un componente de azar (aleatoriedad). Esto significa que si corremos el mismo programa dos veces sobre el mismo grafo, podríamos obtener resultados distintos. Por esta razón, correr una sola prueba no nos daría la imagen completa de la calidad del algoritmo, es por esto que para obtener datos más confiables, ejecutamos cada instancia 10 veces utilizando semillas distintas. De este modo, podemos analizar los resultados desde varios ángulos:

- **Costo promedio:** Nos da una idea del resultado que podemos esperar normalmente.
- **Mejor y Peor caso:** Nos muestra qué tan estable es el algoritmo ante el azar.

- **Tiempo de ejecución:** Permite medir cuánto esfuerzo computacional requiere el algoritmo a medida que el grafo se hace más grande.

SOLUCIONES EXACTAS

Con el objetivo de establecer una base comparativa sólida y comprender el comportamiento del problema ante diferentes niveles de optimización, se implementaron dos algoritmos exactos. Este enfoque permite contrastar una estrategia de búsqueda exhaustiva rudimentaria frente a una versión un poco más optimizada, proporcionando un punto de referencia (*baseline*) antes de incursionar en métodos aproximados y metaheurísticos.

Exacta básica (*branching*)

Dado que el MVC pertenece a la clase NP-completa, la búsqueda del óptimo global requiere explorar el espacio de soluciones de manera exhaustiva. Siendo así, la primera solución implementada consiste en un algoritmo de ramificación recursiva basado en la propiedad fundamental de las aristas: para toda arista $\{u, v\} \in E$, cualquier cobertura válida debe contener, al menos, a uno de sus extremos.

El algoritmo opera seleccionando una arista arbitraria y explorando de forma recursiva dos ramas mutuamente excluyentes:

1. La inclusión del vértice u en el *cover*.
2. La inclusión del vértice v en el *cover*.

En cada rama, se eliminan el vértice elegido y todas sus aristas incidentes, simplificando el grafo hasta alcanzar el caso base (un grafo sin aristas). Donde, si bien este método garantiza la optimalidad al explorar todas las combinaciones factibles, su complejidad temporal es exponencial, lo que limita su aplicación a grafos de pequeña escala.

Pseudocódigo básico:

```
Branching_MVC(G):
  if E(G) está vacío:
    return {}
  escoger una arista (u, v)
  A = Branching_MVC(G sin u) U {u}
  B = Branching_MVC(G sin v) U {v}
  return el conjunto con menor tamaño entre A y B
```

Análisis de complejidad y garantías

La búsqueda explora un árbol binario de decisiones, por lo que el tiempo es exponencial en el tamaño de la solución. En términos generales, el peor caso es exponencial en n, y nuestra implementación base no aplica optimizaciones adicionales. Aun así, el algoritmo garantiza el óptimo global al explorar exhaustivamente ambas ramas, pero a costa de un tiempo de ejecución exagerado y a veces inviable para grafos grandes, densos o complejos. De hecho no se pudo correr en ninguna de las instancias DIMACS seleccionadas.

Exacta mejorada (branch-and-bound)

Para intentar mitigar la explosión combinatoria del método anterior, se desarrolló una versión mejorada que incorpora técnicas de *Branch-and-Bound*, es decir, de ramificación con acotamiento, o simplemente de *poda*. Esta variante busca 'podar' o descartar ramas del árbol de búsqueda que no tienen potencial de superar a la mejor solución hallada hasta el momento, donde las optimizaciones clave incluyen:

- **Reglas de reducción proactivas:** En nuestro caso se implementó la regla de grado 1 (si un nodo tiene grado 1, su vecino debe formar parte del *cover* necesariamente para minimizar el conjunto) y la eliminación de nodos aislados, reduciendo el tamaño del grafo antes de cada ramificación.

- **Estimación de cota inferior (*Lower Bound*):** Se utilizó el tamaño de un *matching* maximal como cota inferior. Dado que para cubrir un *matching* de tamaño k se requieren al menos k vértices, si la suma del *cover* actual más esta cota supera a la mejor solución conocida (*best-so-far*), la rama se descarta inmediatamente, ya que esto significa que no puede conducir a una solución óptima porque sería peor que la mejor encontrada hasta el momento.

- **Cota superior inicial:** A diferencia de la versión básica, este algoritmo inicia con una cota superior obtenida mediante una heurística voraz, lo que permite realizar podas agresivas desde los primeros niveles del árbol.

Esta versión mantiene la garantía de exactitud, pero optimiza drásticamente el tiempo de cómputo al reducir el número de nodos explorados en el árbol de decisión, sin embargo, sigue siendo exponencial en el peor caso.

Pseudocódigo (branch-and-bound):

```
BB_MVC(G, C, best):
 aplicar reducciones (aislados, grado 1) a G y actualizar C
 if E(G) está vacío:
  return C
 if |C| >= |best|: podar
 lb = tamaño de matching maximal en G
 if |C| + lb >= |best|: podar
 escoger arista (u, v)
 sol_u = BB_MVC(G sin u, C U {u}, best)
 actualizar best si sol_u mejora
 sol_v = BB_MVC(G sin v, C U {v}, best)
 actualizar best si sol_v mejora
 return best
```

Análisis de complejidad y garantías

La poda reduce el número de ramas exploradas, pero el peor caso sigue siendo exponencial. La garantía de optimalidad se mantiene porque las podas solo eliminan ramas que no pueden mejorar la mejor solución conocida; por lo tanto, sigue siendo un algoritmo exacto, aunque costoso en grafos grandes o densos.

HEURÍSTICA ESPECIALIZADA: ISOLATION ALGORITHM

A diferencia de las aproximaciones convencionales que priorizan nodos de alta conectividad, el *Isolation Algorithm* (IA) se fundamenta en una premisa de exclusión lógica: si un nodo posee grado 1, su único vecino necesariamente debe formar parte de la cobertura para satisfacer la arista que los une. Esta observación, aunque simple, permite construir una heurística eficiente y efectiva para el problema de Minimum Vertex Cover (MVC) [10].

En consecuencia, la heurística va seleccionando vértices de **grado mínimo** (siempre mayor que cero), agrega a todos sus vecinos al cover y elimina de inmediato esos vecinos junto con las aristas incidentes. La idea es ir "aislando" el grafo, repitiendo el proceso hasta que no queden aristas; por ello puede describirse como un enfoque voraz inverso.

Este procedimiento NO garantiza optimalidad: al escoger nodos de grado mínimo mayores que 1, es posible agregar vecinos que cubren menos aristas de las que podrían cubrir otros candidatos de mayor grado, lo cual puede producir soluciones subóptimas. Sin embargo, en la práctica ofrece resultados razonables en poco tiempo y suele ser más rápido que la heurística de Grado Máximo (que se basa en agregar a la cobertura los nodos de mayor grado), esto, porque en la de Grado Máximo se agrega a la cobertura y elimina del grafo un nodo en cada iteración, mientras que en el algoritmo de aislamiento se agregan a la cobertura y se eliminan del grafo grupos de nodos en cada iteración.

Podemos mencionar adicionalmente que una mejora sencilla consiste en eliminar *nodos redundantes* dentro del cover obtenido: si todos los vecinos de un vértice ya están en la cobertura, dicho vértice no aporta nuevas aristas cubiertas y puede removerse sin perder factibilidad. En nuestra implementación, una vez finalizado el ciclo principal, se escanean los nodos de la cobertura para depurar esos casos.

Pseudocódigo (Isolation Algorithm):

```
Isolation_MVC(G):
 C = {}
 mientras |E(G)| > 0:
  grados = obtener_grados(G)
  escoger v con grado mínimo (> 0)
  C = C U vecinos(v)
  eliminar de G a vecinos(v) y sus aristas incidentes
 escanear C y eliminar nodos redundantes
 retornar C
```

Análisis de complejidad y garantías

Desde el punto de vista computacional, en el peor caso (por ejemplo, un grafo camino) la selección iterativa del nodo de menor grado y la eliminación de sus vecinos se repite aproximadamente $|V|/2$ veces, porque en cada iteración se "salta" al menos dos vértices del camino al incorporar el vecino y removerlo junto con sus aristas incidentes.

En cada paso, encontrar el vértice de grado mínimo y actualizar el grafo requiere recorrer los grados y eliminar nodos, lo cual cuesta $O(|V|)$ en una implementación directa; al repetir este proceso $O(|V|)$ veces se obtiene un costo total $O(|V|^2)$. Por otro lado, la limpieza de redundantes requiere revisar la cobertura y sus adyacencias, lo que implica $O(|V|+|E|)$. Por tanto, la

complejidad temporal global se mantiene en $O(|V|^2)$, mientras que la complejidad espacial es $O(|V|+|E|)$ al trabajar sobre una copia del grafo.

Un ejemplo ilustrativo es un camino de 10 nodos: al escoger un extremo (grado 1), se agrega su vecino al cover y se elimina del grafo; el proceso avanza "saltando" nodos hasta cubrir todas las aristas, produciendo un cover como $\{2,4,6,8,10\}$. Este caso hace visible por qué el número de iteraciones es aproximadamente $|V|/2$. En contraste, en un grafo completo la primera iteración agrega $|V|-1$ vecinos al cover y elimina todas las aristas, de modo que el algoritmo termina en una sola iteración. Esto muestra que el "peor caso" para las operaciones de agregar/eliminar vecinos no coincide con el peor caso global de iteraciones del algoritmo.

En términos de garantías, el Isolation Algorithm no asegura encontrar la solución óptima para el problema MVC, ya que su enfoque voraz puede llevar a elecciones subóptimas. Sin embargo, proporciona una solución factible en tiempo polinómico, siendo útil como punto de partida para métodos más sofisticados como la búsqueda local o metaheurísticas.

BÚSQUEDA LOCAL Y ESTRUCTURA DE VECINDAD

Para abordar MVC con búsqueda local definimos una vecindad basada en intercambios de dos etapas (*two-stage exchange*). El espacio de soluciones se modeló como subconjuntos $C \subseteq V$ y la búsqueda permite estados intermedios no necesariamente factibles, reparándolos de forma local. Esta estrategia evita explorar todos los k-swaps completos, ya que la adición y remoción se realizan en pasos separados.

La búsqueda comienza construyendo una solución inicial con una heurística voraz de máximo grado (se agregan vértices de mayor grado hasta cubrir todas las aristas), luego realiza una remoción inicial para explorar soluciones más pequeñas y, a partir de allí, alterna entre cubrir aristas descubiertas y eliminar vértices con alta ganancia. Adicionalmente, para evitar estancamientos, se incorporan pesos dinámicos en las aristas descubiertas y un mecanismo de olvido controlado por ρ, en otras palabras, se incrementan los pesos de las aristas no cubiertas para guiar la búsqueda hacia soluciones que las incluyan, y periódicamente se reducen estos pesos para evitar que penalizaciones antiguas dominen la trayectoria de búsqueda.

Estructura de vecindad

La vecindad se construye a partir de una arista descubierta elegida aleatoriamente. En cada paso se agrega al cover uno de sus extremos (el que maximiza la función de ganancia) y luego se remueve del cover el vértice con mayor función de ganancia. Este intercambio en dos etapas permite recorrer soluciones cercanas sin enumerar todos los movimientos posibles y mantiene una lista dinámica de aristas descubiertas para actualizar de forma eficiente la factibilidad.

Función de evaluación

La función de evaluación es la función de ganancia, que estima el cambio en el peso total de aristas cubiertas al invertir el estado de un vértice (entrar o salir del cover). Formalmente, la función de ganancia considera solo aristas cuyos extremos no están simultáneamente en el cover; si el vértice está en el cover y su vecino no lo está, su eliminación reduce el peso cubierto, mientras que si el vértice está fuera y su vecino tampoco está, su inclusión incrementa el peso cubierto. Esta métrica guía tanto la elección del vértice a añadir (reparación) como el vértice a remover (intensificación).

Diversificación con pesos de aristas

Cuando una arista queda descubierta, su peso se incrementa para penalizar su ausencia y sesgar la búsqueda hacia vértices que cubran aristas problemáticas. Periódicamente, los pesos se reducen mediante un factor de *olvido* ρ para evitar que penalizaciones antiguas dominen la trayectoria de búsqueda.

Pseudocódigo (búsqueda local):

```
LocalSearch_MVC(G):
 C = cover ávido por máximo grado
 inicializar pesos de aristas a 1
 mientras no se exceda el límite:
  si no hay aristas descubiertas:
   guardar mejor solución y remover un vértice con mayor ganancia
  si hay aristas descubiertas:
   escoger una arista descubierta
   agregar el extremo con mayor ganancia
   remover del cover el vértice con mayor ganancia
   incrementar pesos de aristas descubiertas
   aplicar olvido periódico (factor rho)
 retornar mejor cover
```

Análisis de complejidad y garantías

La búsqueda local no garantiza optimalidad, pero encuentra soluciones de buena calidad en tiempos razonables. Su complejidad depende del número de iteraciones y de las operaciones locales (actualización de pesos y evaluación de la función de ganancia), lo que en la práctica resulta eficiente para grafos medianos y grandes.

METAHEURÍSTICAS - ILS Y GLS

Búsqueda Local Iterada (ILS)

La idea central de ILS es alternar intensificación y diversificación: se explota una búsqueda local hasta un óptimo, se perturba la solución para escapar del estancamiento y se vuelve a intensificar. Esta lógica lo hace especialmente atractivo en nuestro contexto porque reutiliza la búsqueda local ya implementada, mantiene simplicidad de diseño y ofrece un mecanismo claro para explorar nuevas regiones del espacio de soluciones. Por ello se priorizó ILS frente a una búsqueda guiada como metaheurística principal, reservando GLS como contraste experimental.

Siendo así, la solución inicial se construye con un procedimiento voraz (*initial cover*) y luego se mejora con la búsqueda local anterior. A partir de ahí, cada iteración aplica una perturbación que remueve aleatoriamente k vértices del cover actual, donde k se calcula como una fracción del tamaño de la solución (`perturb_fraction`) con un mínimo (`perturb_min`). Este mínimo fija una cantidad base de vértices a remover para garantizar que siempre haya perturbación, aun cuando el cover es pequeño o la fracción produce un valor menor que 1. Cabe acotar que la solución resultante puede ser infactible, por lo que se repara agregando vértices para cubrir

aristas descubiertas mediante una selección ávida: se elige una arista no cubierta al azar y se agrega el extremo de mayor grado (desempate aleatorio). Luego se aplica nuevamente la búsqueda local para llegar a un nuevo óptimo local.

Como decisiones de diseño, incorporamos una memoria de soluciones recientes (hashes de covers) para evitar ciclos. Si un candidato ya aparece en la memoria, aplicamos una perturbación fuerte removiendo aproximadamente la mitad del cover antes de reparar y volver a intensificar. El criterio de aceptación es principalmente elitista (se acepta si mejora el mejor conocido), pero también permite aceptar soluciones de calidad similar con una probabilidad pequeña (`accept_equal_prob`) para fomentar exploración.

Pseudocódigo (ILS):

```
ILS_MVC(G):
 C = cover inicial voraz
 C = LocalSearch(G, C)
 best = C
 memoria = {hash(C)}
 mientras no se exceda el límite:
  k = max(perturb_min, round(perturb_fraction * |C|))
  C' = perturbar_removiendo_k(C, k)
  C' = reparar_cover(G, C')
  C' = LocalSearch(G, C')
  si hash(C') en memoria:
   C' = perturbar_removiendo_k(C', |C'|/2)
   C' = reparar_cover(G, C')
   C' = LocalSearch(G, C')
  aceptar C' (mejora o con prob. accept_equal_prob)
  actualizar best y memoria
 retornar best
```

Análisis de complejidad y garantías

ILS no garantiza optimalidad; hereda las garantías de la búsqueda local y busca escapar de óptimos locales mediante perturbaciones. Su costo está dominado por el número de iteraciones y por las llamadas a la búsqueda local: si T_{LS} es el tiempo de una ejecución de búsqueda local, el costo total es aproximadamente $O(\texttt{max\_iter} \cdot T_{LS})$, más el costo lineal de perturbación, reparación y gestión de memoria. En la práctica, el límite de tiempo controla la ejecución, es decir, se detiene cuando se alcanza `time_limit`, y el mejor cover factible encontrado se devuelve como solución final.

Búsqueda Local Guiada (GLS)

Por curiosidad experimental también implementamos una búsqueda local guiada. Donde ahora la idea es penalizar características problemáticas y optimizar un costo guiado que combine el tamaño del cover con esas penalizaciones. En nuestra implementación, las características son las aristas descubiertas y la función objetivo guiada es:

$$f(C) = |C| + \lambda \sum_{e \in E \text{ no cubierta}} \pi_e,$$

donde π_e es la penalización de la arista e y λ es un parámetro de peso.

El procedimiento parte de una solución inicial voraz y mantiene penalizaciones iniciales en cero. En cada iteración, ejecuta una búsqueda local guiada que recorre los vértices del cover en orden aleatorio e intenta removerlos; la remoción se acepta únicamente si la solución sigue siendo factible y reduce el costo guiado. Luego se identifican las aristas descubiertas y se incrementan sus penalizaciones. De esta forma, las aristas que permanecen sin cubrir aumentan su influencia en el costo y empujan la búsqueda a configuraciones distintas, evitando insistir en los mismos patrones. No se usan perturbaciones explícitas: la diversificación proviene del ajuste dinámico de penalizaciones y el mejor cover factible encontrado se mantiene como solución final.

Pseudocódigo (GLS):

```
GLS_MVC(G):
 C = cover inicial voraz
 penalizaciones pi_e = 0 para toda arista e
 best = C
 mientras no se exceda el límite:
  C = GuidedLocalSearch(G, C, pi, lambda)
  uncovered = aristas no cubiertas por C
  para e en uncovered:
   pi_e = pi_e + 1
  si |C| < |best|: best = C
 retornar best

GuidedLocalSearch(G, C, pi, lambda):
 improved = True
 mientras improved:
  improved = False
  mezclar orden de vértices en C
  para v en C:
   C' = C \ {v}
   si C' es factible y f(C') < f(C):
    C = C'
    improved = True
    romper
 retornar C
```

Análisis de complejidad y garantías

GLS no garantiza optimalidad; su objetivo es mejorar soluciones locales mediante un costo guiado que penaliza aristas descubiertas, y el costo por iteración está dominado por la búsqueda local guiada, que evalúa remociones y verifica factibilidad sobre las aristas; además, se recorre el conjunto de aristas descubiertas para actualizar penalizaciones. Mientras que en la práctica, el tiempo se controla con `max_iter` o `time_limit`, y el mejor cover factible observado se mantiene como salida.

RESULTADOS EXPERIMENTALES

En esta sección reportamos resultados cuantitativos para las instancias DIMACS descritas en el marco experimental. Se ejecutaron cuatro algoritmos: heurística de aislamiento (IA), búsqueda

local, ILS y GLS. Todos los resultados reportan el **costo promedio** (tamaño del cover), el **gap** respecto al óptimo conocido y el **tiempo promedio** por corrida.

Configuración experimental

- **Plataforma:** Linux
- **Lenguaje/entorno:** Python 3.12.3 en entorno virtual.
- **Parámetros generales:** `max_iter`=10000, `lambda_penalty`=0.3.
- **Semillas:** diferentes por corrida para capturar aleatoriedad.
- **Nota sobre `time_limit`:** el límite controla el bucle externo de cada algoritmo. En ILS/GLS se invoca búsqueda local dentro de la iteración, por lo que el tiempo total por corrida puede exceder el valor nominal de `time_limit`.
- **Protocolos comparados:**
  - **Principal:** `time_limit`=1s y `num_runs`=10.
  - **Sensibilidad:** `time_limit`=1s con `num_runs`{5,25} y `time_limit`=2s con `num_runs`{5,10}.

Resultados por instancia (time_limit=1s, num_runs=10)

Cuadro 0.2: Costo promedio y gap respecto al óptimo conocido.

Instancia	Ópt.	H	Gap	LS	Gap	ILS	Gap	GLS	Gap
C125.9	91	94	3.0	91	0.0	93.4	2.4	95.7	4.7
DSJC500.5	487	489	2.0	490	3.0	489.5	2.5	490.9	3.9
MANN_a27	252	253	1.0	253	1.0	253.3	1.3	260.8	8.8
brock200_2	188	191	3.0	191	3.0	190.8	2.8	192.2	4.2
brock400_2	371	377	6.0	377	6.0	378.4	7.4	382.3	11.3
brock800_2	776	782	6.0	783	7.0	782.8	6.8	785.5	9.5
gen200_p0.9_44	156	163	7.0	167	11.0	164.3	8.3	168.3	12.3
hamming10-4	984	988	4.0	996	12.0	993.3	9.3	1000.1	16.1
hamming8-4	240	240	0.0	240	0.0	240.7	0.7	246.9	6.9
keller4	160	160	0.0	162	2.0	162.0	2.0	163.8	3.8
keller5	749	752	3.0	760	11.0	758.3	9.3	760.5	11.5
p_hat300-1	292	293	1.0	293	1.0	293.1	1.1	294.3	2.3

Cuadro 0.3: Tiempo promedio por corrida (segundos).

Instancia	H	LS	ILS	GLS
C125.9	0.002	1.005	2.703	1.002
DSJC500.5	0.082	1.188	2.542	1.451
MANN_a27	0.009	1.030	2.304	1.003
brock200_2	0.011	1.027	2.340	1.031
brock400_2	0.031	1.074	2.320	1.120
brock800_2	0.167	1.431	3.108	2.559
gen200_p0.9_44	0.004	1.013	2.206	1.005
hamming10-4	0.127	1.486	3.042	2.468
hamming8-4	0.013	1.039	2.351	1.040
keller4	0.006	1.016	2.115	1.012
keller5	0.107	1.310	2.693	1.875
p_hat300-1	0.039	1.086	2.684	1.158

Cuadro 0.4: Resumen agregado (12 instancias, time_limit=1s, num_runs=10).

Algoritmo	Gap prom.	Gap min	Gap max	Tiempo prom. (s)
Heurística (IA)	3.00	0.0	7.0	0.050
Búsqueda local	4.75	0.0	12.0	1.142
ILS	4.49	0.7	9.3	2.534
GLS	7.94	2.3	16.1	1.394

Hallazgos y análisis por algoritmo

- **Heurística (IA).** Es la más rápida (tiempos del orden de milisegundos) y completamente determinista: en todas las instancias el mejor y el peor caso coinciden. Logra óptimo en *hamming8-4* y *keller4* y gaps pequeños en grafos medianos. Su debilidad aparece en grafos densos o más grandes (*gen200_p0.9_44*, *brock* y *hamming10-4*), con gaps entre 4 y 7, lo cual era esperable al ser una heurística voraz.

- **Búsqueda local.** Mantiene tiempos estables alrededor de 1–1.5s y obtiene soluciones óptimas en *C125.9* y *hamming8-4*. Sin embargo, en instancias densas el gap crece de forma clara (hasta 12 en *hamming10-4* y 11 en *gen200_p0.9_44* y *keller5*). Al operar bajo límite de tiempo fijo, los resultados son estables (variabilidad prácticamente nula), pero no siempre mejoran a la heurística en instancias difíciles, es decir, la búsqueda local puede estancarse en óptimos locales subóptimos.

- **ILS.** Presenta el mejor equilibrio entre calidad y tiempo. En grafos densos reduce el gap de la búsqueda local (por ejemplo, *gen200_p0.9_44*: 11.0 $\rightarrow$ 8.3; *hamming10-4*: 12.0 $\rightarrow$ 9.3; *keller5*: 11.0 $\rightarrow$ 9.3), con un costo temporal moderado (~2.5s promedio), consistente con el uso de búsqueda local dentro del ciclo. En instancias fáciles no siempre supera a la búsqueda local (*C125.9*), lo que sugiere que la perturbación introduce ligeras pérdidas cuando el óptimo local ya es muy bueno.

- **GLS.** Es más lento que la búsqueda local y sistemáticamente más débil en calidad (gap promedio 7.94). Su esquema de penalizaciones no logró compensar la pérdida por removals conservadores, sobre todo en grafos densos donde el gap sube a 16.1 (*hamming10-4*) y 12.3 (*gen200_p0.9_44*). En este conjunto de parámetros, GLS actúa como contraste experimental pero no supera ILS ni la búsqueda local.

Sensibilidad a parámetros

- **Número de corridas.** Al comparar `num_runs`=5, 10 y 25 (con `time_limit`=1s), los costos promedio cambian marginalmente. Por ejemplo, ILS en *C125.9* pasa de 93.4 (5 corridas) a 93.4 (10) y 93.24 (25); GLS en *hamming10-4* oscila entre 1002.6 (5), 1000.1 (10) y 1002.04 (25). Esto indica que 10 corridas son suficientes para estimar el promedio con baja varianza.

- **Límite de tiempo.** Al duplicar `time_limit` de 1s a 2s, el tiempo promedio casi se duplica en búsqueda local, ILS y GLS, pero la mejora en calidad es pequeña o nula. Ejemplos: ILS en *C125.9* mantiene gap 2.4 con 1s y 2s; GLS en *DSJC500.5* conserva gap 3.9; y búsqueda local en *hamming10-4* permanece en gap 12.0. Hay mejoras puntuales (p.ej. búsqueda local en *keller5* pasa de gap 11.0 a 9.0), pero no son consistentes. En consecuencia, `time_limit`=1s ofrece la mejor relación costo/beneficio para esta entrega.

Experimento adicional: ILS/GLS con `time_limit`=5s

Para evaluar si más tiempo mejora realmente la calidad para las metaheurísticas, ejecutamos ILS y GLS con un `time_limit` de 5s (10 corridas) y comparamos contra ILS y GLS con 1s. En ILS, la mejora es clara en instancias densas: *hamming10-4* pasa de gap 9.3 (1s) a 6.4 (5s) y *keller5* de 9.3 a 6.9. En *brock800_2* la mejora es ligera (6.8 $\rightarrow$ 6.7), pero en *brock400_2* el gap empeora (7.4 se mantiene en 7.4, sin mejoras visibles) y en instancias fáciles como *C125.9* o *p_hat300-1*

prácticamente no hay cambio. En términos de tiempo, el costo computacional aumenta sustancialmente (~8–13s promedio en ILS), por lo que la ganancia en calidad es localizada, no general.

En GLS con 5s el comportamiento es muy similar al de 1s: los gaps se mantienen prácticamente iguales (p.ej., *hamming10-4*: 16.1 $\rightarrow$ 16.1; *brock400_2*: 11.3 $\rightarrow$ 11.3), y no se observa una mejora consistente. En síntesis, el aumento de tiempo beneficia a ILS solo en algunas instancias densas, mientras que GLS no muestra mejoras relevantes respecto a 1s bajo estos parámetros.

CONCLUSIONES

Los resultados confirman patrones esperables en MVC: la heurística IA es extremadamente rápida y produce soluciones razonables, pero pierde precisión en instancias densas; la búsqueda local mejora la calidad con tiempos estables; e ILS ofrece el mejor balance calidad/tiempo, con mejoras claras en grafos densos. GLS, en cambio, no supera a ILS con los parámetros actuales y actúa como contraste experimental.

El análisis de sensibilidad sugiere que aumentar el número de corridas estabiliza los promedios, mientras que incrementar el tiempo aporta mejoras puntuales pero no sistemáticas. En conjunto, los resultados son consistentes con lo esperado para MVC: técnicas más sofisticadas mejoran la calidad a costo computacional moderado.

En términos metodológicos, el uso de instancias DIMACS permite comparar de manera objetiva el gap con óptimos conocidos y evaluar el compromiso entre calidad y tiempo. Además, la diferencia entre heurísticas rápidas y esquemas de intensificación/diversificación muestra que el rendimiento depende fuertemente de la densidad del grafo y de la capacidad del método para escapar de óptimos locales.

Como trabajo futuro, sería útil explorar estrategias de diversificación más agresivas en GLS, ajustar parámetros de perturbación en ILS y estudiar límites de tiempo globales más estrictos para una comparación uniforme. También es razonable incorporar métricas adicionales (varianza y distribuciones de costos) para caracterizar con más detalle la estabilidad de las soluciones.

REFERENCIAS

[1] R. M. Karp. *Reducibility among combinatorial problems*. En: *Complexity of Computer Computations*. 1972.

[2] M. R. Garey y D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. 1979.

[3] V. Vazirani. *Approximation Algorithms*. Springer-Verlag, 2001.

[4] J. A. Bondy y U. S. R. Murty. *Graph Theory*. Springer, 2008.

[5] R. Diestel. *Graph Theory* (5th ed.). Springer, 2017.

[6] D. B. West. *Introduction to Graph Theory*. Prentice Hall, 2001.

[7] W. Pullan. *Phased local search for the minimum vertex cover problem*. Journal of Combinatorial Optimization, 2009.

[8] F. N. Abu-Khzam et al. *Kernelization algorithms for the vertex cover problem: Theory and experiments*. ALENEX, 2006.

[9] P. Bhawalkar et al. *Influence maximization in social networks*. International Colloquium on Automata, Languages, and Programming, 2012.

[10] O. Ugurlu. *New Heuristic Algorithm for Unweighted Minimum Vertex Cover*. Proceedings of the International Conference on Problems of Cybernetics and Informatics (PCI), 2012. Disponible en: https://pci.cyber.az/2012/papers/v4/23.pdf

[11] H. Khattab, A. Sharieh y B. A. Mahafzah. *Most Valuable Player Algorithm for Solving Minimum Vertex Cover Problem*. International Journal of Advanced Computer Science and Applications (IJACSA), Vol. 10, No. 8, 2019.