Diseño y Análisis de Algoritmos

Problema 2: Estrategia de Batallones en la guerra

Integrantes:

- Ana Karla Caballero González
- Carlos Arturo Pérez Cabrera
- Diana Laura Pérez Trujillo

1. Problema

En medio de una gran guerra, los generales están planeando una estrategia con sus batallones. Cada batallón controla un área específica del territorio, y algunos de estos territorios pueden superponerse, lo que crea conflictos de control entre los batallones.

Dada una colección finita C de territorios controlados por los batallones, y un número entero positivo $k \leq |C|$, tu misión es determinar si es posible organizar al menos k batallones de manera que cada uno controle un territorio completamente distinto de los demás, es decir, sin superposiciones ni conflictos entre ellos.

2. Modelación Matemática

Sea $G = (V, E)$ un grafo no dirigido donde $v \in V(G)$ representa un territorio de un batallón y donde $(u, v) \in E(G)$ existe solo si entre u y v existe un conflicto, es decir los territorios se superponen.

3. Análisis NP-Completitud

En este problema nos piden determinar si es posible ubicar k batallones tal que estos no entren en conflicto por compartir territorio con otro batallón.
El problema se puede resumir a encontrar un conjunto independiente de tamaño k, pues implicaría que se pueden ubicar k batallones y que ningún par de estos compartan territorio.
Demostraremos que este problema es NP:

Teniendo el grafo con los batallones ubicados es fácil verificar si la instancia del problema realmente cumple con la característica deseada en tiempo polinómico, iterando uno por uno los vértices verificando si ningún vértice tiene grado ≥ 0, esto implicaría que hay conflictos entre batallones. Esta verificación se hace en $O(n)$ donde n es la cantidad de batallones ubicados, por lo tanto se puede hacer en tiempo polinomial, entonces es NP.
Demostremos que es NP-Completo:
Primero es necesario hacer una reducción a un problema que sea NP-completo. Para ello, se utilizará el Problema del conjunto independiente, que consiste en comprobar si un grafo dado tiene un conjunto independiente de un tamaño k. Es equivalente reducir al problema del clique de tamaño k (en cuyo caso habría que trabajar con el grafo complemento del deseado). En cualquiera de los dos enfoques, es directamente equivalente al problema presentado, por tanto, se puede afirmar que el problema de los batallones es NP completo.

4. Soluciones

En todas las soluciones propuestas, se trabajó con G' grafo complemento de G. La motivación principal para ello fue que encontrar un conjunto independiente de tamaño k es equivalente a encontrar un clique de tamaño k en el complemento del grafo.

Una anotación importante es hacer un análisis de las componentes conexas en el Grafo. Si en el grafo G original, existe una cantidad de componentes conexas mayor a k, entonces se puede afirmar que se puede lograr una distribución de k batallones sin que ocurran conflictos, pues sería asociar cada uno al territorio de su componente conexa.

4.1. Primera Solución

La primera solución propuesta consiste en un algoritmo de "fuerza bruta". Primero se organizan los nodos de mayor a menor por sus grados y se seleccionan k primeros nodos de la lista. De no cumplir la condición deseada, se selecciona alguno otro, siempre el de mayor grado posible de los que quedan. Si se encuentra un clique de tamaño k, entonces se termina el algoritmo.
Con esta solución se revisan todos los nodos y es posible determinar si existe en el grafo un clique de tamaño k.

4.1.1. Complejidad Temporal

Ordenar los nodos es $O(nlog(n))$. Para seleccionar los k nodos y tratar de encontrar un clique de tamaño k, en el peor caso se generan todas las combinaciones posibles, además por cada nodo de los k que se quieran incluir, es necesario verificar si son adyacentes a los restantes. En conclusión, este algoritmo resulta extremadamente ineficiente siendo su complejidad temporal $O(n!/(k!(n-k)!))$.

4.2. Segunda Solución

Para la segunda propuesta se investigó acerca de la existencia de alguna Metaheurística que permitiera mejorar el tiempo y ejecución de la solución.
Finalmente, se seleccionó utilizar Colonia de Hormigas. Esta técnica se inspira en el comportamiento de las hormigas en la naturaleza, que utilizan feromonas para comunicarse y guiarse hacia fuentes de alimento. El mecanismo de exploración que propone esta metaheurística supone un conjunto de k hormigas que se mueven por un grafo que representa el problema, construyendo soluciones de manera independiente e iterativa. Luego que cada hormiga termina, se ajustan los valores de feromona de la etapa y se vuelve a realizar este mismo proceso por varias iteraciones. Logrando que mediante varias hormigas artificiales se creen soluciones a un problema de optimización. Si bien este mecanismo es el encargado de la convergencia del algoritmo, se infiere que puede ser lento si resolvemos problemas de mediana o alta dimensión.

En el problema en particular de decidir si se puede encontrar un clique de tamaño k, las hormigas pueden ser vistas como exploradores que recorren el grafo, construyendo soluciones parciales al seleccionar nodos que forman parte del clique. A medida que las hormigas encuentran cliques válidos, depositan feromonas en los nodos y aristas involucrados, lo que aumenta la probabilidad de que otras hormigas sigan rutas similares en futuras iteraciones. Este mecanismo permite a la metaheurística concentrarse en áreas prometedoras del espacio de soluciones.

Para la implementación se diseñó una clase Ant_Colony que encapsulara el comportamiento de la colonia de hormigas deseado.
Recordar que se estará trabajando con el Grafo complemento del grafo de entrada, pues el problema original consiste en decidir si se puede hallar un conjunto independiente de tamaño k, lo que es equivalente a decidir si se puede hallar un clique de tamaño k en el grafo complemento de G.

Inicialmente, se establece el mismo nivel de feromonas en cada arista del grafo. Se cuenta con la implementación de un método para el calculo de un valor simple que representa la probabilidad de seleccionar un nodo basado en la cantidad de feromonas en sus aristas.
Además, se define un método para la creación del clique de la siguiente manera:

- Cada hormiga comienza desde un nodo aleatorio y trata de agregar nodos al clique actual.
- Se seleccionan nodos posibles basándose en la probabilidad ponderada por las feromonas.
- Se verifica si el nodo seleccionado es vecino de todos los nodos en el clique actual antes de agregarlo.
- Si se encuentra un clique del tamaño requerido, se devuelve inmediatamente.

Después de cada intento de construcción del clique, se refuerzan las feromonas en las aristas que forman parte del clique encontrado. Luego, se aplica una tasa de evaporación a todas las feromonas en el grafo para simular la pérdida natural de feromonas con el tiempo.
Luego de una cantidad de iteraciones prefijadas, si se encuentra un clique que cumple con el tamaño mínimo, se devuelve su tamaño y los nodos que lo componen. Si no se encuentra ningún clique después de todas las iteraciones, devuelve 0 y una lista vacía.

4.2.1. Complejidad Temporal

Se utiliza una solución que está acotada por el problema del clique. En el peor de los casos se tendrá que revisar todos los vértices, por lo que la complejidad temporal final del algoritmo sería $O(k * \binom{|V|}{k})$.

5. Casos prueba

Para esta solución se implementó un generador de casos prueba. En primer lugar, se define una función $generate_graph$ que crea un grafo aleatorio con un número especificado de nodos y aristas. Esta función genera nodos y establece conexiones entre ellos, asegurándose de que no haya bucles ni conexiones duplicadas. Posteriormente, en la función main, se configuran parámetros como el número de nodos, aristas, hormigas, y otros parámetros del algoritmo, antes de instanciar la clase AntColony con el grafo generado. Finalmente, se invoca el método $solve()$ para que el algoritmo busque soluciones óptimas en el grafo. Este enfoque permite probar y evaluar el rendimiento del algoritmo de colonia de hormigas en diferentes configuraciones de grafos aleatorios. Luego, se chequea si es correcta la existencia de dicho clique.

6. Organización y requerimientos

En la carpeta raíz del proyecto se encuentra una distribución para cada problema. En la carpeta $Problema2$ se encuentra el archivo $problema2.tex$. En la carpeta *code* se encuentran los archivos de código fundamentales: $Ant\_Colony.py$ que contiene la implementación de la colonia de hormigas, $Brute_Force.py$ que contiene la primera solución presentada, $Complement$ un archivo auxiliar para hallar el complemento de un grafo, $Generator.py$ que contiene el generador de casos prueba, $Graph.py$ con la implementación de grafo que se ha diseñado y los archivos $Test.py$ y $Check.py$ que se utilizan para realizar el proceso de testing.

Todo fue implementado en el lenguaje de programación $Python$.