Problemas P y NP

1. Definición de un Problema P y de una NP ¿Son iguales, donde se diferencian?

Los problemas P y NP son mejor conocidos como Clases P y Clases NP, la Clase P significa que el tiempo que se necesita para dar una solución a un problema que pertenece a esta clase es polinomial o en el peor caso $O(n^k)$ donde k es una constante, por otro lado, la clase NP significa nondeterministic polynomial-time en inglés, si lo quisiéramos traducir al español sería algo como *"no necesariamente tiempo polinomial"*

A grandes rasgos, un problema de decisión pertenece a la clase P si podemos encontrar una solución del problema que sea fácil. Un problema de decisión es aquel que podemos contestar con una respuesta de sí o no. La mayoría de los problemas que nos encontramos no son de esta forma, sino que son de optimización. Aun así, si encontramos una solución que sea de decisión, en la mayoría de los casos podemos adaptarla a un problema de optimización.

Un problema de decisión pertenece a NP si la comprobación de la solución es fácil. Uno de los problemas más conocidos que pertenecen a NP, es el "Problema del Agente Viajero". Lo que nos pregunta es lo siguiente: si un agente de ventas debe visitar n ciudades que están interconectadas, ¿es posible que las visite todas recorriendo una distancia menor a d? Para poder encontrar la solución, debemos realizar una búsqueda exhaustiva entre todas las posibles combinaciones hasta que demos con ella. Pero si ya la tenemos, solamente debemos comprobar que la distancia es menor a d y que pasa por las n ciudades.

Supongamos que cambiamos la pregunta por, ¿es cierto que no puede visitar las n ciudades en una distancia menor a d? Encontrar la respuesta es igual de difícil que la pregunta anterior, sin embargo, comprobar la respuesta no es análogo al problema anterior. Si contamos con la respuesta, aun así, debemos realizar todas las comprobaciones para verificar que la respuesta es correcta. Por lo tanto, esta variación de la pregunta ya no es NP, dentro de la categoría NP existe la subcategoría NP-completo de los cuales detallaremos más adelante.

Ahora literalmente hablando la pregunta del millón de dorales (digo literalmente ya que es una pregunta que aún no tiene solución y aquel que de la solución recibirá un millón de dólares), ahora la pregunta es la siguiente **¿P=NP?**

Es decir, se tiene que demostrar que si existe un problema en NP-c ∩ P, entonces P=NP, NP-c se refiere a NP-completo, se toman estos problemas ya que son los más difíciles de solucionar y esto nos ayuda ya que si se llega a comprobar que NP-c ∩ P entonces todo NP=P.

Entonces podemos tener los siguientes hechos:

1.- $Si\ \pi \in NP - c \cap P, existe\ un\ algoritmo\ polinomial\ que\ resuelve\ \pi, por\ estar\ \pi\ en\ P.$

$Por\ otro\ lado, como\ \pi\ es\ NP - completo, para\ todo\ \pi' \in NP, \pi' \preccurlyeq \pi.$

Nota: El símbolo $\preccurlyeq$ se define para la reducción polinomial.

2.- $Sea\ \pi' \in NP.\ Apliquemos\ la\ reducción\ polinomial\ que\ transforma\ instancias\ de\ \pi'\ en\ \ instancias\ de\ \pi\ y\ luego\ el\ algoritmo\ polinomial\ que\ resuelve\ \pi. Por\ definición\ de\ reduccion\ polinomial, es\ facil\ ver\ que\ lo\ que\ se\ obtiene\ es\ un\ algoritmo\ polinomial\ que\ resuelve\ \pi'.$

Hasta el momento no se conoce ningún problema en NP-c ∩ P, es decir, un problema que cumple con ser NP-c y ser P, así como tampoco se ha demostrado que un problema esté en NP\P, es decir, lo contrario que sea NP, pero no P, en el caso de probar que NP\P se probaría que P≠NP.

En la Imagen 1 del lado derecho podemos ver el caso si P≠NP fuera verdadera, del lado izquierdo podemos ver el caso si P=NP.

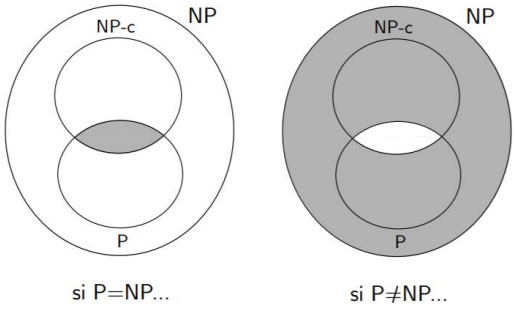

Imagen 1. Casos si P=NP o P≠NP.

2. Problemas NP-completos o clase NP-completos

Dentro de la categoría NP existe la subcategoría NP-completo. Decimos que un problema es NP-completo si pertenece a NP y cualquier otro problema en NP puede ser transformado a este en tiempo polinomial. Para poder decir que un problema es NP-completo, sólo debe ser equivalente a otro y automáticamente lo será a los demás que ya están demostrados. El problema inicial que se usó para hacer la comparación es conocido como "Problema de la Satisfacibilidad Booleana" y mediante el Teorema de Cook sabemos que será equivalente a los demás NP-completo.

Otra categoría es la conocida como NP-difícil (o NP-duro), que a pesar del nombre no es una subcategoría de NP. Un problema es considerado NP-difícil si, suponiendo que contamos con alguna forma para resolverlo en tiempo constante, podemos utilizar estas soluciones para resolver un problema NP-completo en

tiempo lineal. Por esto, se dice que un problema es NP-difícil si es tanto o más complicado que uno NP-completo, independientemente de si pertenece a NP o no.

Si un problema de optimización tiene una contraparte de decisión que pertenece a NP-completo, entonces este problema de decisión es NP-difícil. Por ejemplo, optimizar el problema del agente viajero (encontrar la menor distancia d para visitar todos los pueblos) es NP-difícil.

En otras palabras, menos formales un problema NP-completo son los más difíciles de todos los problemas NP.

3 y 4.- Tiempo polinomial y Verificación del tiempo polinomial

Un algoritmo de tiempo polinomial se define como aquel con función orden **O(p(n))** para alguna función polinómica p, donde n denota el tamaño de la entrada. Los algoritmos de tiempo exponencial, $\boldsymbol{O(e^n)}$, son los que el número de ciclos que tienen que realizarse con el algoritmo es proporcional a la función $\boldsymbol{e^n}$ de modo que el poder computacional necesario para correr el algoritmo crece de forma exponencial al tamaño n del problema.

La mayoría de los algoritmos de tiempo exponencial son simples variaciones de una búsqueda exhaustiva, mientras que los algoritmos de tiempo polinomial, usualmente se obtienen mediante un análisis más profundo de la estructura del problema. En la teoría de la complejidad computacional, existe el hecho de que un problema no está "bien resuelto" hasta que se conozca un algoritmo de tiempo polinomial que lo resuelva. Por tanto, nos referiremos a un problema como intratable, si es tan difícil que no existe algoritmo de tiempo polinomial capaz de resolverlo.

5.- Reducción NP-completos

La definición formal del NP-completo emplea reducciones o transformaciones de un problema a otro. Si queremos resolver el problema P, pero ya conocemos la solución para cierto problema Q. Entonces, sea T una función que tome una entrada x para P y produzca una T(x), que sea una entrada de Q tal que la respuesta correcta para P con x es Sí, sí y sólo si la respuesta en Q con T(x) también es Sí.

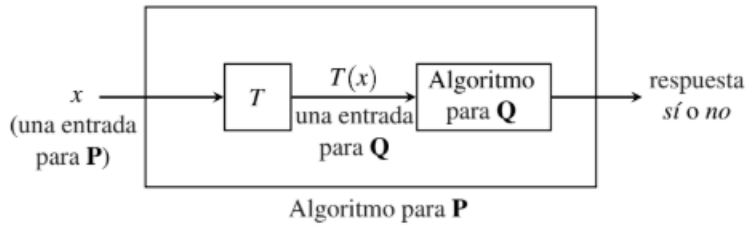

Figura 13.2 Reducción de un problema **P** a un problema **Q**: la respuesta del problema **Q** con $T(x)$ debe ser la misma que la respuesta de **P** con x.

La reducción en general es muy importante. Por ejemplo, si tenemos funciones de una librería para resolver cierto problema y si nosotros podemos reducir un nuevo

problema a uno de los problemas resueltos, ahorramos mucho tiempo. Consideremos el ejemplo de un problema en el que tenemos que encontrar la ruta mínima del producto en un grafo dirigido dado, donde el producto de la ruta es la multiplicación de los pesos de los bordes a lo largo de la ruta. Si tenemos un código para el algoritmo de Dijkstra para encontrar la ruta más corta, podemos tomar el registro de todos los pesos y usar el algoritmo de Dijkstra para encontrar la ruta mínima del producto en lugar de escribir un código nuevo para este nuevo problema.

Nota: En esta sección nos referimos a P como un problema, no a la Clase P.

6.- Demostración de NP-complejos

A partir de la definición de NP-completo, parece imposible demostrar que un problema P es NP-completo. Por definición, requiere que demostremos que cada problema en NP es un tiempo polinómico reducible a P. Afortunadamente, hay una forma alternativa de probarlo y es lo que se habló anteriormente si existe una transformación T, que pase el problema P a Q, entonces P es NP-completo, esta idea es la que generalmente se usa para demostrar que cierto problema P es NP-completo.

¿Cuál fue el primer problema probado como NP-completo?

Debe haber algún primer problema NP-completo probado por definición de problemas NP-completo. SAT (problema de satisfacción booleana) es el primer problema NP-completo probado por Cook.

Siempre es útil saber sobre si un problema es NP-completo incluso para ingenieros. Supongamos que se pide escribir un algoritmo eficiente para resolver un problema extremadamente importante para una empresa. Después de mucho pensar, solo puede llegar a un enfoque de tiempo exponencial que no es práctico. Si no se sabe acerca de NP-completo, se puede decir que no hay un algoritmo eficiente. Si se conoce el concepto de NP-completo y se demuestra que el problema es NP-completo, se puede decir con orgullo que es poco probable que exista la solución de tiempo polinomial. Si hay una posible solución de tiempo polinomial, entonces esa solución resuelve un gran problema de la informática que se mencionó al inicio.

Ahora para ser mas interesante esta parte vemos como Cook probo que el problema SAT es un problema NP-completo.

Definamos primero el problema SAT: El problema SAT consiste en decidir si, dada una formula lógica φ expresada como conjunción de disyunciones (ej.: $\varphi = x_1 \wedge (x_2 \vee \neg x_1) \wedge (x_3 \vee \neg x_4 \vee x_1)$), existe una valuación de sus variables que haga verdadera φ. Es fácil ver que SAT $\in$ NP. El certificado en este caso sería una valuación que satisfaga φ. Evaluar una fórmula es polinomial.

Teorema de Cook: SAT es NP-completo:

La demostración de Cook es directa: Se considera un problema genérico $\pi \in NP$ y una instancia genérica $d \in D_\pi$. A partir de la hipotética NDTM (Máquinas de Turing No Determinísticas) que resuelve π, genera en tiempo polinomial una fórmula lógica $\varphi_{\pi,d}$ en forma normal (conjunción de disyunciones) tal que $d \in Y_\pi$ sí y sólo si $\varphi_{\pi,d}$ es satisfactible.

A partir del Teorema de Cook, la técnica estándar para probar que un problema π es NP-completo aprovecha la transitividad de $\preccurlyeq$, y consiste en lo siguiente:

1. Mostrar que π esta en NP.
2. Elegir un problema π' apropiado que se sepa que es NP-completo.
3. Construir una reducción polinomial f de π' en π.

La segunda condición en la definición de problema NP-completo sale usando la transitividad: Sea π'' un problema cualquiera de NP. Como π' es NP-completo, $\pi'' \preccurlyeq \pi'$. Como probamos que $\pi' \preccurlyeq \pi'$, resulta $\pi'' \preccurlyeq \pi$.

Nota: En esta sección nos referimos a P como un problema, no a la Clase P y el símbolo $\preccurlyeq$ se define para la reducción polinomial.

7.- Problemas NP-completos

Para esta parte daremos una lista de 21 problemas computacionales famosos, que tratan sobre combinatoria y teoría de grafos, ademas cumplen la característica en común de que todos ellos pertenecen a la clase de complejidad de los NP-completos. Esta lista fue elaborada en 1972 por el informático Richard Karp, en su trabajo seminal "Reducibility Amone Combinatorial Problems" o en español *"Reducibilidad entre Problemas Combinatorios"*, como profundización del trabajo de Stephen Cook de quien se habló anteriormente y fue el que demostró el problema SAT.

Mientras que la pertenencia del problema SAT a la clase de los NP-completos fue demostrada utilizando mecanismos particulares, las pertenencias de los 21 problemas siguientes fueron demostradas mediante reducciones polinomiales. Así, el problema SAT se redujo polinomialmente a los problemas 0-1 INTEGER PROGRAMMING, CLIQUE y 3-SAT, y estos a su vez se redujeron a otros varios. La lista completa es la que se muestra a continuación.

- SAT (Problema de satisfacibilidad booleana, para fórmulas en forma normal conjuntiva)
    - 0-1 INTEGER PROGRAMMING (Problema de la programación lineal entera)
    - CLIQUE (Problema del clique)
        - SET PACKING (Problema del empaquetamiento de conjuntos)
        - VERTEX COVER (Problema de la cobertura de vértices)
            - SET COVERING (Problema del conjunto de cobertura)

- FEEDBACK NODE SET
- FEEDBACK ARC SET
- DIRECTED HAMILTONIAN CIRCUIT (Problema del circuito hamiltoniano dirigido)
    - UNDIRECTED HAMILTONIAN CIRCUIT (Problema del circuito hamiltoniano no dirigido)

- 3-SAT (Problema de satisfacibilidad booleana de 3 variables por cláusula)
    - CHROMATIC NUMBER (Problema de la coloración de grafos)
        - CLIQUE COVER (Problema de la cobertura de cliques)
        - EXACT COVER (Problema de la cobertura exacta)
            - HITTING SET
            - STEINER TREE
            - 3-DIMENSIONAL MATCHING (Problema del matching tridimensional)
            - KNAPSACK (Problema de la mochila)
                - JOB SEQUENCING (Problema de las secuencias de trabajo)
                - PARTITION (Problema de la partición)
                    - MAX-CUT (Problema del corte máximo)

Tras un tiempo se descubrió que muchos de estos problemas podían ser resueltos si su enunciado se particularizaba a unas ciertas clases, o podían ser resueltos aproximadamente con un error máximo de un cierto porcentaje. Sin embargo, David Zuckerman demostró en 1996 que cada uno de estos 21 problemas tiene una versión restringida de optimización que es no aproximable a menos que P = NP, demostrando que la versión de la reducción.

8.- Planteamiento de 10 problemas con su algoritmo

1) Partition Problem (Problema de la partición):

En teoría de números y la informática, el problema de la partición, o el número de partición, es la tarea de decidir si un determinado conjunto múltiple S de enteros positivos puede ser dividido en dos subconjuntos S1 y S2 tal que la suma de los números en S1 iguales la suma de los números en S2 . Aunque el problema de la partición es NP-completo , hay un tiempo pseudo-polinomial de programación dinámica, y hay heurística que resuelven el problema en muchos casos, ya sea de manera óptima o aproximadamente. Por esta razón, se le ha llamado "el más sencillo problema NP-duro".

Pseudocódigo usando programación Dinámica:

```
Entrada:  Una lista de números enteros S
Salida: Verdadero si S puede ser particionado en dos y esas dos partes
tienen el mismo valor de la suma de sus elementos.
```

```
1 function find_partition(S):
2     n = S.length
3     K = sum(S)
4     P = [K/2+1][n+1] //Tipo boolean
5     inicializar la primera fila (P(0,x)) de P como True
6     inicializar la primera columna(P(x, 0)) de P como True, excepto
P(0, 0
7     for i from 1 to [K/2]
8         for j from 1 to n
9             P(i, j) = P(i, j-1)
10            if i >= S(j – 1)
11                P(i, j) = P (i, j) or P(i – S(j – 1),j – 1)
12    return P([K/2], n)
```

El código se adjunta en el correo, Este algoritmo se ejecuta en tiempo O(K*N), donde N es el número de elementos en el conjunto de entrada y K es la suma de los elementos en el conjunto de entrada.

2) CHROMATIC NUMBER (Problema de la coloración de grafos)

No encontré un algoritmo como tal, solo de forma hablada:

1-Colorea el primer vértice con el primer color.

2-Haga lo siguiente para los vértices V-1 restantes

Considere el vértice seleccionado actualmente coloréalo con el color con el número más bajo que no se ha usado en vértices previamente coloreados adyacente a este si todos los colores utilizados anteriormente aparecen en vértices adyacentes a v, asígnele un nuevo color.

El código se adjunta al correo, la complejidad es de O(V^2 + E) en el peor caso, donde v es el número de vértices del grafo

3) UNDIRECTED HAMILTONIAN CIRCUIT (Problema del circuito hamiltoniano no dirigido)

La ruta hamiltoniana en un grafo no dirigido es una ruta que visita cada vértice exactamente una vez. Un ciclo hamiltoniano (o circuito hamiltoniano) es un camino hamiltoniano tal que hay un borde (en el gráfico) desde el último vértice hasta el primer vértice del camino hamiltoniano. El pseudocodigo es el siguiente:

```
Entrada:  Un grafo de matriz 2D [V][V] donde V es el número de
vértices en el grafo y el grafo[V][V] es la representación de matriz
de adyacencia del grafo. Un grafo de valores [i][j] es 1 si hay un
borde directo de i a j; de lo contrario, el gráfico [i] [j] es 0.
```

```
Salida: Una ruta de matriz [V] que debe contener la ruta de Hamilton.
[i] debería representar el i-ésimo vértice en la ruta de Hamiltoniana.
El código también debe devolver falso si no hay un ciclo Hamiltoniano
en el gráfico.
El algoritmo se divide en dos partes, veamos la primera parte
Entrada: Vértice v y posición K
Salida: Comprueba si colocar v en la posición k es válido o no.
1 function esValido(v, k):
2     if no hay borde entre el nodo (k-1) a v
3         return false
4     if v ya está tomado, entonces
5         return false
6     return true//De otra manera es verdadera

Veamos la segunda parte
Entrada: Nodo del grafo
Salida: Regresa true cuando hay un ciclo hamiltoniano, de lo contrario
regresa false.
1 function cicloEncontrado(node k):
2     if todos los nodos están incluidos
3         if hay un borde entre los nodos k y 0
4             return true
5         else
6             return false
7     for i from 1 to V
8         if esValido(v,k)
9             add v al camino
10            if cicloEncontrado(k+1)
11                return true
12            remove v del camino
13    return false
```

El código se adjunta al correo, la complejidad es de O((v-1)!) en el peor caso, donde v es el número de vértices del grafo.

4) Subset Sum Problem

Dado un conjunto de enteros no negativos y una suma de valor, determine si hay un subconjunto del conjunto dado con una suma igual a la suma dada.

Para resolver este problema tiempo pseudo-polinomial, se utiliza la programación dinámica, cuyo Pseudocodigo es el siguiente.

```
Entrada: Arreglo con los valores y suma objetivo
Salida: Regresa true cuando se puede obtener la suma objetivo dado los
valores
1 function subsetSum(int set[], int sum):
2     int n = set.length
3     boolean subset[][] = new boolean[sum + 1][n + 1];
4     for i from 1 to sum
5         for j from 1 to n
6             subset[i][j] = subset[i][j - 1]
7             if (i >= set[j - 1])
8                 subset[i][j] = subset[i][j]
                                  || subset[i - set[j - 1]][j - 1]
9    return subset[sum][n]
```

La complejidad de este algoritmo es O(sum*n), donde sum es la suma que queremos obtener y n es el tamaño del arreglo. El código se adjunta al correo

5) SAT (Problema de satisfacción booleana)

La satisfacción booleana o simplemente SAT es el problema de determinar si una fórmula booleana es satisfactoria o insatisfactoria.

Satisfactible: si las variables booleanas se les pueden asignar valores tales que la fórmula resulte VERDADERA, entonces decimos que la fórmula es satisfactoria.

Insatisfactorio: si no es posible asignar tales valores, entonces decimos que la fórmula no es satisfactoria.

No hay algoritmo que solucione este problema para una formula de tamaño n, pero existe una versión 2-SAT que si es programable

2-SAT es un caso especial de problema de satisfacción booleana y puede resolverse en tiempo polinomial.

Para entender esto mejor, primero veamos qué es la Forma Conjuntiva Normal (CNF) o también conocida como Producto de Sumas (POS).

CNF: CNF es una conjunción (AND) de cláusulas, donde cada cláusula es una disyunción (OR).

Ahora, 2-SAT limita el problema de SAT a solo aquellas fórmulas booleanas que se expresan como un CNF con cada cláusula que tiene solo 2 términos (también llamado 2-CNF).

Ahora el código en C++:

```
int n;
vector<vector<int>> g, gt;
vector<bool> used;
vector<int> order, comp;
vector<bool> assignment;

void dfs1(int v) {
    used[v] = true;
    for (int u : g[v]) {
        if (!used[u])
            dfs1(u);
    }
    order.push_back(v);
}

void dfs2(int v, int cl) {
    comp[v] = cl;
    for (int u : gt[v]) {
        if (comp[u] == -1)
            dfs2(u, cl);
    }
}

bool solve_2SAT() {
    used.assign(n, false);
    for (int i = 0; i < n; ++i) {
        if (!used[i])
            dfs1(i);
    }

    comp.assign(n, -1);
    for (int i = 0, j = 0; i < n; ++i) {
        int v = order[n - i - 1];
        if (comp[v] == -1)
            dfs2(v, j++);
    }

    assignment.assign(n / 2, false);
    for (int i = 0; i < n; i += 2) {
        if (comp[i] == comp[i + 1])
            return false;
        assignment[i / 2] = comp[i] > comp[i + 1];
    }
    return true;
}
```

Ahora podemos implementar todo el algoritmo. Primero construimos el grafo y encontramos todos los componentes fuertemente conectados. Esto se puede lograr con el algoritmo de Kosaraju en tiempo **O (n+m).** En el segundo recorrido del grafo,

el algoritmo de Kosaraju visita los componentes fuertemente conectados en orden topológico.

6) Aproximación de la cubierta de vértices

Encontrar la cubierta de vértices dado un grafo, es un problema NP-completo,

```
Entrada: Un grafo
Salida: Cobertura de vértice correspondiente
1 function coberturaVertice(grafo G = (V,E)):
2     while(E!=ø):
3         tomar una esquina arbitraria (u,v)≤E
4         añadir ambos, u y v a la cubierta del vertice
5         borrar todas las esquinas de E que sean incidente a u o v
```

7) Operación OR de booleanos

Dado un conjunto de booleanos, en ese conjunto ¿hay al menos uno de ellos tiene el valor de verdadero?, como existe la transformación de true -> 1, false -> 0, entonces, esto, es un problema NP-completo:

```
Entrada: Conjunto[n] tipo booleano
Salida: Booleano
1 function opOR(Conjunto):
2     int n = Conjunto.length
3     bool resultado = false
4     for i from 0 to n
5         if(Conjunto[i]==true):
6             resultado = true
7             break
8     return resultado
```

8) Operación AND de booleanos

Similar al punto 7 pero ahora es con AND, ya que el anterior es NP-completo, este también.

```
Entrada: Conjunto[n] tipo booleano
Salida: Booleano
1 function opAND(Conjunto):
2     int n = Conjunto.length
3     bool resultado = true
```

```
4     for i from 0 to n
5         if(Conjunto[i]==false):
6             resultado = false
7             break
8     return resultado
```

9) Travelling salesman problem (TSP) o el problema del viajero

Este es uno de los problemas más conocidos, y a menudo se lo llama un problema difícil. Un vendedor debe visitar n ciudades, pasando por cada ciudad solo una vez, comenzando desde una de ellas que se considera su base y regresando a ella. se proporciona el costo del transporte entre las ciudades (cualquier combinación posible). Se solicita el programa del viaje, que es el orden de visitar las ciudades de tal manera que el costo sea el mínimo.

Numeremos las ciudades de 1 a n, y dejemos que la ciudad 1 sea la base de la ciudad del vendedor. También supongamos que c (i, j) es el costo de visita de i a j. Puede haber c (i, j) <> c (j, i). Aparentemente todas las soluciones posibles son (n-1)! Alguien podría determinarlas sistemáticamente, encontrar el costo de cada una de estas soluciones y finalmente quedarse con la que tiene el costo mínimo. requiere al menos (n-1)! pasos.

Un algoritmo heurístico de acuerdo con este algoritmo cada vez que el vendedor está en la ciudad i elige su próxima ciudad, la ciudad j para la cual el costo c (i, j) es el mínimo entre todos los costos c (i, k), donde k son los punteros de la ciudad que el vendedor aún no ha visitado. También existe una regla simple en caso de que más de una ciudad proporcione el costo mínimo, por ejemplo, en tal caso, se elegirá la ciudad con la k más pequeña. Este es un algoritmo codicioso que selecciona en cada paso la visita más barata y no le importa si esto conducirá a un resultado incorrecto o no.

Ahora el pseudocodigo:

```
Entrada: Numero de ciudades n y el arreglo con los costos respectivos
(se empieza desde la ciudad numero 1)
Salida: Vector con el recorrido y el costo total
1 function TSP(n,c):
2     C=0
3     costo=0
4     recorrido = 0
5     e=1
6     for r from 1 to n-1
7         elegir el puntero j con
```

```
8          mínimo=c(e,j)=min{c(e,k)}
9          costo = costo + mínimo
10         e = j
11         C[r] = j
12         C[n]=1
13         costo= costo +c(e,1)
```

Podemos encontrar situaciones en las que el algoritmo TSP no da la mejor solución. También podemos tener éxito en mejorar el algoritmo. Por ejemplo, podemos aplicar el algoritmo t veces para t ciudades diferentes y mantener la mejor ronda cada vez. También podemos deshacer la codicia de tal manera que se reduzca el problema del algoritmo, es decir, no hay espacio para elegir los lados al final del algoritmo porque los lados más baratos se han agotado.

10) Dominating set (Conjunto dominante)

En la teoría de grafos, un conjunto dominante para un gráfico G = (V, E) es un subconjunto D de V de tal manera que cada vértice que no está en D es adyacente a al menos un miembro de D. El número de dominación es el número de vértices en un conjunto dominante más pequeño para G.

Se cree que puede no haber un algoritmo eficiente que encuentre un conjunto dominante más pequeño para todos los gráficos, pero existen algoritmos de aproximación eficientes.

En este caso se presenta un algoritmo de aproximación

Algoritmo:

1. Primero tenemos que inicializar un conjunto "S" como vacío
2. Tome cualquier borde "e" del gráfico que conecta los vértices (digamos A y B)
3. Agregue un vértice entre A y B (digamos A) a nuestro conjunto S
4. Eliminar todos los bordes en el gráfico conectado a A
5. Vuelva al paso 2 y repita, si todavía queda algo de borde en el gráfico
6. El conjunto final S es un conjunto dominante del gráfico

Pseudocodigo:

```
Antes que nada, declarar Vector<Integer> []g, boolean []box y g se
llenara de la siguiente manera:
g = new Vector[ver];
    for (int i = 0; i < ver; i++)
        g[i] = new Vector<Integer>();
Donde ver es el número de vértices.
Entrada: Numero de vértices y aristas del grafo
```

```
Salida: Conjunto dominante
1 function Dominant(vertices, aristas):
2     S <- Vector
3     for i from 1 to n-1
4         if(!box[i])
5             S.add(i)
6             box[i]=true
7             for j from 0 to g[i].size()
8                 if(!box[g[i].get(j)])
9                       box[g[i].get(j)] = true
10                      break;
11     return S
```

El código se adjunta al correo.

Fue algo complicada investigar y encontrar los pseudocodigos para estos ejemplos, pero sin duda es un tema muy interesante para investigar.

Nota: Todos los códigos fueron programados en Java y tiene ya entradas predefinidas, prácticamente todos son clases main con excepción de la clase Grado la cual es un objeto.

Referencias:

Libros:

Baase, S., Van Gelder, A., Escalona García, R. and Torres, S., 2002. Algoritmos Computacionales. 3rd ed. México: Pearson Educación.

Cormen, T., Leiserson, C., Rivest, R. and Stein, C., 2014. Introduction To Algorithms. 3rd ed. Cambridge, Massachusetts: MIT Press.

Skiena, S., 2009. The Algorithm Design Manual. 2nd ed. Dordrecht: Springer.

Páginas Web:

[1] Universidad de Buenos Aires (2006, marzo 05). Clases de complejidad computacional: P y NP. [Online]. Available: https://www.dm.uba.ar/materias/optimizacion/2006/1/fbonomo_clase_npc.pdf

[2] GeeksforGeeks. (2011, junio 17). What does ‘Space Complexity’ mean? [Online]. Available: https://www.geeksforgeeks.org/g-fact-86/?ref=lbp

[3] G.H. Pier (sin fecha). 2.6. Clases P (Polynomial) y NP (Nondeterministic Polynomial). [Online]. Available: https://pier.guillen.com.mx/algorithms/02-analisis/02.6-clases.htm

[4] Wikipedia (2020, junio 12). NP-completeness. [Online]. Available: https://en.wikipedia.org/wiki/NP-completeness

[5] Wikipedia (2016, febrero 12). Clases de complejidad P y NP. [Online]. Available: https://es.wikipedia.org/wiki/Clases_de_complejidad_P_y_NP

[6] Wikipedia. (2020, mayo 16). Teoría de la complejidad computacional. [Online]. Available: https://es.wikipedia.org/wiki/Teor%C3%ADa_de_la_complejidad_computacional

[7] Wikipedia. (2019, septiembre 20). Lista de 21 problemas NP-completos de Karp. [Online]. Available: https://es.wikipedia.org/wiki/Lista_de_21_problemas_NP-completos_de_Karp

[8] Wiki. (Sin fecha). problema de la partición - Partition problema. [Online]. Available: https://es.qwe.wiki/wiki/Partition_problem

[9] GeeksforGeeks. (2013, noviembre 14). Graph Coloring | Set 2 (Greedy Algorithm). [Online]. Available: https://www.geeksforgeeks.org/graph-coloring-set-2-greedy-algorithm/

[10] GeeksforGeeks. (2012, diciembre 24). Subset Sum Problem | DP-25. [Online]. Available: https://www.geeksforgeeks.org/subset-sum-problem-dp-25/

[11] GeeksforGeeks. (2017, julio 09). 2-Satisfiability (2-SAT) Problem. [Online]. Available: https://www.geeksforgeeks.org/2-satisfiability-2-sat-problem/

[12] GeeksforGeeks. (2018, agosto 19). Dominant Set of a Graph. [Online]. Available: https://www.geeksforgeeks.org/dominant-set-of-a-graph/

[13] Sin autor. (Sin fecha). The T.S.P. Example. [Online]. Available: http://students.ceid.upatras.gr/~papagel/project/tspprobl.htm

[14] S.Christine. (Sin fecha). Hamiltonian Cycle. [Online]. Available: https://www.tutorialspoint.com/Hamiltonian-Cycle

Videos:

[1] Derivando (2017, abril 10). ¿Qué es eso del problema P versus NP? [Online]. Available: https://www.youtube.com/watch?v=UR2oDYZ-Sao

[2] Abdul Bari (2018, febrero 28). 8. NP-Hard and NP-Complete Problems. [Online]. Available: https://www.youtube.com/watch?v=e2cF8a5aAhE