Contents

1 Stable Maching problema

1.1 Algoritmo Gale-Shapley

Este algoritmo al terminar de ejecutarse se encuentra un matching prefecto si:

- Si existen n solicitantes con diferentes preferencias.
- Si existen n requeridos con diferentes preferencias.

Eligiendo las estructuras correctamente se puede plantear en $O(n)$.

```
Inicialmente M=Vacio

Mientras existe un solicitante sin pareja que no aun se haya
postulado a todas las parejas

    Sea s un solicitante sin pareja
    Sea r el requerido de su mayor preferencia al que no le
                solicito previamente

    if r esta desocupado
        M = M U (s,r)
        s esta ocupado
    else
        Sea s' tal que (s', r) pertenece a M

        si r prefiere a s sobres s'
            M = M - {(s', r)} U (s,r)
            s esta ocupado
            s' esta libre
Retornar M
```

Listing 1: Algoritmo de Gale-Shapley

1.2 Alternativas

1.2.1 Diferentes cantidades de oferentes que requeridos

Dado n oferentes y m requeridos, con $m <> n$, no se puede encontrar un matching stable.

Entonces, tenemos que redefinir el concepto de estable. Una pareja (s,r) es **estable** si:

- No existe requerido r' sin pareja al que s prefiera a su actual pareja.
- No existe un requerido r' en pareja, tal que s y r' se prefieran sobre sus respectivas parejas.
- No existe solicitante s' sin pareja al que r prefiera a su actual pareja.
- No existe un solicitante s' en pareja tal que r y s' se prefieran sobre sus respectivas parejas.

Por lo tanto un matching es estable si:

- No tienen parejas inestables bajo la condicion anterior.
- Que no queden requeridos y solicitantes sin pareja.

Soluciones para ajustar al modelo de Gale-Shapley:

1. Inventar $|n - m|$ elementos ficticios
    - Los elementos ficticios se pondran en las listas de preferencias con menos elementos.
    - Estos elementos ficticios se agregan al final y deben ser los menos preferidos.
    - Luego ejecutar Gale-Shapley
    - Por ultimo, eliminar las parejas con elementos ficticios. Estos seran los requeridos que quedan sin pareja.
2. Adecuar el Algoritmo

- Si hay mas **solicitantes** que requeridos, quitar de la *lista de solicitantes* sin parejas a aquellos que agotaron sus propuestas.
- Si hay mas **requeridos** que solicitantes, quitar de la *lista de parejas* a aquellas donde el requerido quedo sin pareja.

1.2.2 Preferencias incompletas

Las listas de preferencias de los oferentes y los requeridos son un subset de las contrapartes.

Son parejas **aceptables** de un elemento a aquellas contrapartes que figuran en su lista de preferencias.

Una pareja (s,r) es **estable** si:

- Son *aceptables* entre ellos.
- No existe requerido *aceptable* r' sin pareja al que s prefiera a su actual pareja.
- No existe un requerido *aceptable* r' en pareja, tal que s y r' se prefieran sobre sus respectivas parejas.
- No existe solicitante *aceptable* s' sin pareja al que r prefiera a su actual pareja.
- No existe un solicitante *aceptable* s' en pareja tal que r y s' se prefieran sobre sus respectivas parejas.

Un matching es estable si no tiene parejas inestables bajo la condicion anterios.

```
Inicialmente M=Vacio

#Iterea mientras no haya acotado su sublista de preferencias
Mientras existe un solicitante sin pareja
                'que no aun se haya postulado a todas las parejas'

    Sea s un solicitante sin pareja
    Sea r el requerido de su mayor preferencia al que no le
                solicito previamente

    # se condiera si es aceptable
    if r considera 'aceptable' a s

        if r esta desocupado
            M = M U (s,r)
            s esta ocupado
```

```
        else
            Sea s' tal que (s', r) pertenece a M
            si r prefiere a s sobres s'
                M = M - {(s', r)} U (s,r)
                s esta ocupado
                s' esta libre

# Retornar solo parejas aceptables
Retornar M
```

Listing 2: Algoritmo para parejas incompletas

1.2.3 Preferencias con empates

INDIFERENCIA Y PREFERENCIA ESTRICTA

1. X es **indiferente** a "y" y a "z" si en su lista de preferencias estan el la misma posicion.
2. X es **prefefiere estrictamente** a "y" sobre "z" si en su lista de preferencias no le son indiferentes y "y" se encuentra antes que "z" en la misma.

ESTABILIDAD DEBIL

Una pareja (s,r) es debilmente estable si no existe una pareja (s' y r') talque:

- s prefiere estrictamente a r' sobre r *(pareja actual de s)*
- r' prefiere estrictamente a s sobre s' *(pareja actual de r')*

```
    Inicialmente M=Vacio

    #Iterea mientras no haya acotado su sublista de preferencias
    Mientras existe un solicitante sin pareja
                    'que no aun se haya postulado a todas las
    parejas'

        Sea s un solicitante sin pareja
        Sea r el requerido de su mayor preferencia al que no le
                    solicito previamente

        if r esta desocupado
            M = M U (s,r)
```

```
            s esta ocupado
        else
            Sea s' tal que (s', r) pertenece a M

            # prefiere estrictamente
            si r prefiere estrictamente a s sobres s'
                M = M - {(s', r)} U (s,r)
                s esta ocupado
                s' esta libre

    Retornar M
```

Listing 3: Algoritmo para parejas incompletas

En caso de que sea empate, se mantendra con su pareja actual.

ESTABILIDAD FUERTE
Una pareja (s,r) es debilmente estable si no existe una pareja (s' y r') talque:

- s prefiere estrictamente o le es indiferente a r' sobre r *(pareja actual de s)*
- r' prefiere estrictamente o le es indiferente a s sobre s' *(pareja actual de r')*

Puede no existir un matching perfecto.

```
    Inicialmente M=Vacio

    Mientras existe un solicitante sin pareja y no exista
    solicitante que agoto sus parejas

        Sea s un solicitante sin pareja
        Sea r el requerido de su mayor preferencia al que pueda
    proponer
        Por cada sucesor s' a s en la lista de preferencias de r
            if (s',r) pertenece a M
                M = M - {(s',r)}
                s' esta libre
            quitar s' de la lista de preferencias de r
            quitar r de la lista de preferncias de s'

        Por cada requerido r' que tiene multiples parejas
            Por cada pareja s' en pareja con r'
                M = M - {(s',r')}
```

```
17                quitar s' de la lista de preferencias de r'
18                quitar r' de la lista de preferencias de s'
19
20    if estan todos en pareja
21        Retornar M
22    else
23        No existe ningun matching super estable
```

Listing 4: Algoritmo para parejas super estables

En caso de que sea empate, se mantendra con su pareja actual.

1.2.4 Agrupacion de 1 a muchos

El solicitante puede tener varios cupos por lo tanto:

- Exiten m requeridos, donde un requerido puede estar unicamente con 1 pareja.
- Exiten n solicitantes, donde cada solicitante puede tener c cupos para armar parejas.

Existe un matching estable si la cantidad de requeridos es igual a la cantidad de solicitantes por la cantidad de cupos.

$$m = n * c \tag{1}$$

No cambia la definición de Gale Shampey para **matching estable**

```
Inicialmente M=Vacio

Mientras exista un solicitante con cupo disponible

    Sea s un solicitante sin pareja
    Sea r el requerido de su mayor preferencia al que no le
                solicito previamente

    if r esta desocupado
        M = M U (s,r)
        s decremente su disponibilidad de parejas
    else
        Sea s' tal que (s', r) pertenece a M

        si r prefiere a s sobres s'
            M = M - {(s', r)} U (s,r)
            s decremente su disponibilidad de parejas
            s' incrementa su disponibilidad de parejas
Retornar M
```

Listing 5: Algoritmo de solicitantes con cupos

La complejidad algoritmica no se modifica porque solo se agrega un contador.

1.2.5 Agrupacion de muchos a 1

El requerido puede tener varios cupos por lo tanto:

- Exiten m requeridos, donde cada solicitante puede tener z cupos para armar parejas.
- Exiten n solicitantes, donde un requerido puede estar unicamente con 1 pareja.

Existe un matching estable si la cantidad de solicitantes es igual a la cantidad de requeridos por la cantidad de cupos.

$$n = m * z \tag{2}$$

No cambia la definición de Gale Shampey para **matching estable**

```
Inicialmente M=Vacio

Mientras exista un solicitante con cupo disponible

    Sea s un solicitante sin pareja
    Sea r el requerido de su mayor preferencia al que no le
                solicito previamente

    if r tiene cupo
        M = M U (s,r)
        s esta ocupado
        r decrementa su disponibilidad de parejas
    else
        Sea s' tal que (s', r) pertenece a M y
                s' es el menos preferidos de las parejas r

        si r prefiere a s sobres s'
            M = M - {(s', r)} U (s,r)
            s esta ocupado
            s' esta libre
Retornar M
```

Listing 6: Algoritmo de requeridos con cupos

La complejidad algoritmica si se modifica.

Para conocer el solicitante de menor preferencia podemos utilizar un heap de minimos. Como el cupo es de z, la complejidad algoritmica para actualizar el heap es $log(z)$.

1.2.6 Agrupacion de y a x

- Exiten n solicitantes, donde cada solicitante puede tener c cupos para armar parejas.
- Exiten m requeridos, donde cada requerido puede tener z cupos para armar parejas.

Existe un matching estable si:

$$n * c = m * z \tag{3}$$

No cambia la definición de Gale Shampey para **matching estable**
Para implementar se requieren las siguientes estructuras:

- Un heap de minimos para los requeridos.
- Un contador de cupos para los solicitantes.

La complejidad algoritmica es igual a la de los requeridos con cupos

1.2.7 Conjuntos no bipartios - Stable Roommate Problem

Pendiente

2 Analisis amortizado

2.0.1 Metodo de agregacion

2.0.2 Metodo del banquero

2.0.3 Metodo del potencial

2.0.4 Heap binomial y fibonacci

Revisar capitulo 19 del Corven.
Para el **heap binomial** se utilizan bosques de arboles binarios. Existe un proceso donde se van ordenando los arboles.

Al insertar, se parece al ejemplo de contador binario y la amortizacion es O(1)
Decrementar en un log binomial, es log(n) porque no es posible amortizar
Eliminar el minimo, es el el peor caso es log(n)

Para el **heap fibonacci** ...

3 Algoritmos Greedy

Utiliza heurisica de seleccion para encontrar una solución global optima despues de muchos pasos.

3.1 Mochila fraccionaria

Dado un contener de capacidad W, y un conjunto de elementos n fraccionables de valor v_i y peso w_i

El objetivo es seleccionar un subconjunto de elemento o fracciones de ellos de modo de maximizar el valor almacenado y sin superar la capacidad de la mochila.

La complejidad es $O(nlog(n))$

3.2 Cambio de moneda

Es una solución es conocido como solución de cajero. Contamos con un conjunto de diferentes monedas de diferentes denominación sin restricción de cantidad.

$$\$ = (C_1, C_2, C_3, \cdots, C_n)$$

El objetivo es entregar la menor cantidad posible de monedas como cambio.

Tiene una complejidad de $O(n)$.

El sistema \$ se conoce como **canonico** a aquel en el que para todo x, $greedy(\$, x) = optimo(\$, x)$.

Para saber si una base es canonica:

1. Basta con buscar un contraejemplo. Estaria entre la 3ra denomininacion y la suma de las ultimas dos doniminaciones.
2. Utilizar un algoritmo Polinimico para determinar si es un sistema canonico.

Si el problema no es greddy, se puede construir un algoritmo utilizando programación dinamica.

3.3 Interval Scheduling: Algoritmo de Greedy Stay Ahead

Tenemos un conjunto de requests $\{1, 2, .., n\}$; el request i^{th} corresponde a un intervalo de tiempo que comienza al instante $s(i)$ y finaliza al instante $f(i)$. Diremos que un subconjunto de requests es compatible si no hay dos de ellos que al mismo tiempo se superponen, y nuestro objetivo es aceptar un subconjunto compatible tan grande como sea posible. El conjunto compatible con mayor tamaño sera el **óptimo**.

La idea básica en un algoritmo greedy para interval scheduling es usar una simple regla para seleccionar el primer request i_1. Una vez que el request i_1 aceptado, rechazamos todos los request que no son compatibles con i_1. Luego seleccionamos el siguiente request i_2, y volvemos a rechazar todos lo request que no son compatibles con i_2. Continuamos de esta manera hasta que nos quedemos sin requests. El desafio en diseñar un buen algoritmo greedy esta en decidir que regla usar para la selección.

Pueden probar con varias reglas, pero las mas optimo es la siguiente idea: Aceptaremos el request que termina primero, o sea el request para el cual tiene el menor $f(i)$ posible. Asi nos aseguramos que nuestros recursos se liberen tan pronto como sea posible mientras satisfacemos un request. De esta manera podemos maximizar el tiempo restante para satisfacer otro request.

Para escribir el pseudo código, utilizaremos R para denotar al conjunto de request que aún no estan aceptados ni rechazados, y usaremos A para denotar al conjunto de los request aceptados.

```
1  Inicialmente R contiene todos los requests, y A es un conjunto
       vacio.
2
3  Mientras R no esta vacio
4
5      Seleccionar un request i de R que tenga el instante de
       finalizacion mas chico.
6      Agregar el registro i a A
7      Eliminar todos los request de R que no sean compatibles con el
       request i
8
9  Fin mientras
10
11 Retornar el conjunto A como el conjunto de los request aceptados.
```

Listing 7: Algoritmo de greedy para Interval Scheduling

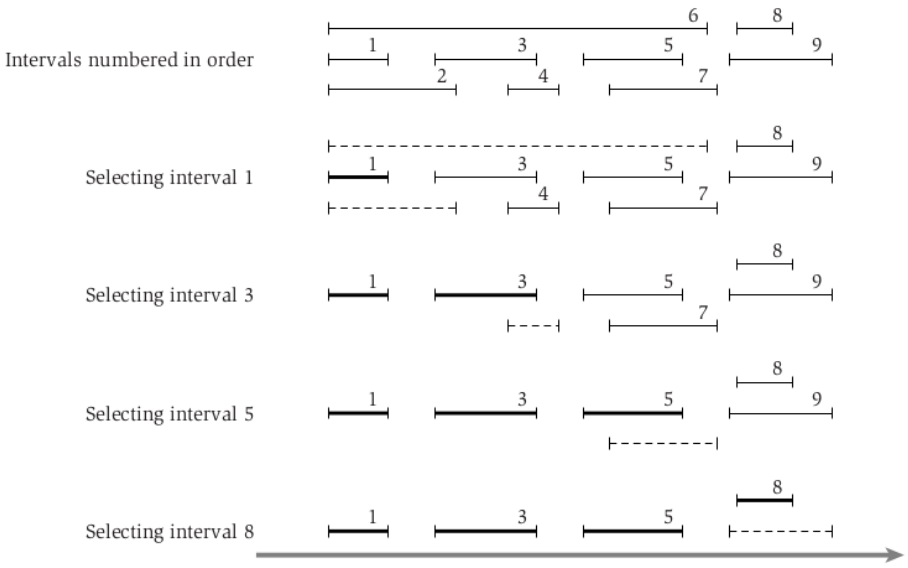


Figure 4.2 Sample run of the Interval Scheduling Algorithm. At each step the selected intervals are darker lines, and the intervals deleted at the corresponding step are indicated with dashed lines.

De forma inmediata podemos decir que el conjunto retornado tiene request compatibles.

Lo que necesitamos es demostrar que la solución es optima. Definimos a O, un conjunto de intervalos optimos. Luego, vamos a mostrar que $|A| = |O|$, o sea que el conjunto A tiene la misma cantidad de intervalos que O, y por lo tanto, A tambien es una solución optima.

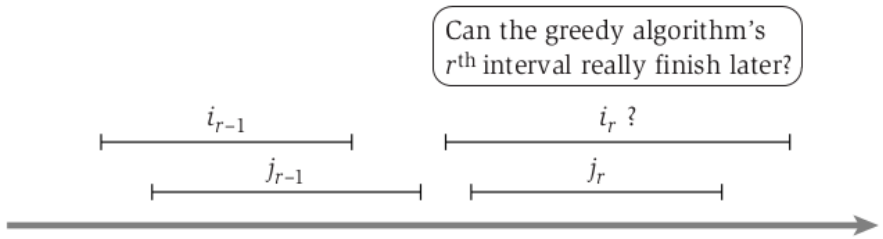
Para la prueba introduciremos la siguiente notación:

- Dado $\{i_1, ..., i_k\}$ el conjunto de request en A en orden que fueron agregados a A. Notar que $|A| = k$.
- Dado $\{j_1, ..., j_m\}$ el conjunto de request en O ordenos de izquierda a derecha. Notar que $|O| = m$.

El objetivo es probar que $k = m$.

La manera en que el algoritmo de greedy se mantenga adelante(**stays ahead**) es que cada uno de sus intervalos finalice al menos tan pronto como lo haga el

correspondiente intervalo en el conjunto O.



(3.1) Para todos los indices $r < k$ tenemos que $f(i_r) \leq f(j_r)$

Demostración: Probaremos la sentencia anterior mediante el método inductivo. Para $r = 1$ la sentencia anterior es cierta, el algoritmo empieza seleccionando el request i_1 con el menor tiempo de finalización.

Para el caso inductivo, o sea $r > 1$ asumiremos como nuestra hipotesis inductiva que la sentencia es verdadera para $r - 1$, y queremos probar que es tambien es lo es para r. La hipotesis inductiva nos dice que asumamos verdadero que $f(i_{r-1}) \leq f(j_{r-1})$. Queremos demostrar que $f(i_r) \leq f(j_r)$.

Dado que O consiste en intervalos compatibles, sabemos que $f(j_{r-1}) \leq s(j_r)$. Combinando esto último con la hipotesis inductiva $f(i_{r-1}) \leq f(j_{r-1})$, obtenemos $f(i_{r-1}) \leq s(j_r)$. Asi el intervalo j_r esta en conjunto R de los intervalos disponibles al mismo tiempo cuando el algoritmo de greedy selecciona i_r. El algoritmo de greedy selecciona el intervalo con el *tiempo final mas chico* (i_r); y dado que intervalo j_r es uno de estos intervalos, tenemos que $f(i_r) \leq f(j_r)$, completando asi el paso inductivo.

De esta forma demostramos que nuestro algoritmo se mantiene adelante del conjunto optimo O. Ahora veremos porque esto implica optimalidad del conjunto A de algoritmo de greedy.

El algoritmo de greedy retorna un conjunto A óptimo.

Demostración: Para demostrarlo utilizaremos la contradicción. Si A no es optimo, entonces el conjunto O debe tener mas requests, o sea que tenemos $m > k$ y aplicando 3.1, cuando r=k, obtenemos que $f(i_k) \leq f(j_k)$. Dado que

$m > k$, existe un request j_{k+1} en O. Este request empieza despues que el request j_k termina y por consiguiente despues de que el request i_k termine. Entonces, despues de eliminar todos los requests que no son compatibles con los request $i_1, ..., i_k$, el conjunto de posibles requests R aún contiene el requests j_{k+1}. Pero el algoritmo de greedy se detiene con el request i_k y este supuestamente se detiene porque R esta vacio, lo cual es una contradicción.

3.4 Seam Carving - TODO

Es un algoritmo para adecuar imagenes. Analiza imagenes recortando pixeles de menor importancia. Retira tantas vetas como sea necesario para llegar a un tamaño optimo.

3.5 Caminimos Minimos - TODO

Dado dos nodos, uno inicial s y otro final t el algoritmo encuentra el camino minimo que los une, tambien entre s y el resto de los nodos.

3.6 Compresión de datos - TODO

El algoritmo de greedy arma un arbol de "hufman" para armar un arbol optimo de prefijos.

4 División y conquista

4.1 Teorema mestro - TODO

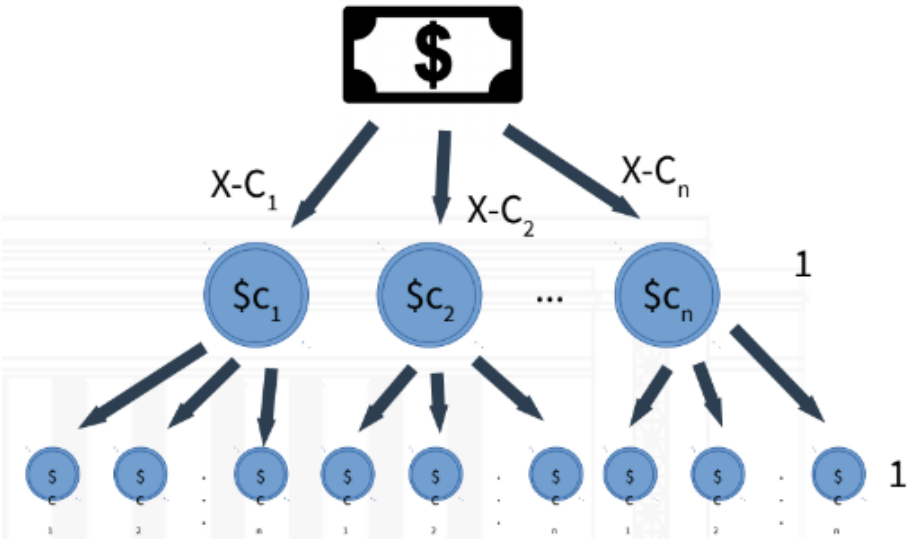
4.2 Mediana con datos separadas

5 Programación dinamica

5.1 Cambio de monedas

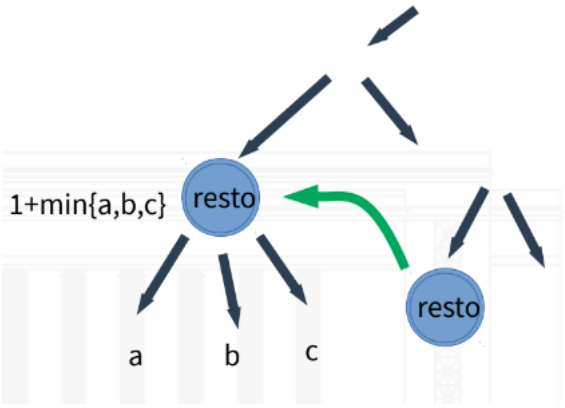
Contamos con un conjunto de monedas de diferente denominación sin restricción de cantidad. Representamos de esta manera $\$ = (c_1, c_2, ..., c_n)$ y tenemos un importe x a dar. Concluimos que no existe un algoritmo satisfactorio de greedy para resolver este problema.

Si buscamos la solución por **fuerza bruta**, se puede armar un arbol de decisión. Por cada moneda posible, se genera un subproblema.



Entonces el camino a la hoja con menor profundidad es la menor cantidad de monedas a dar. Esto hace que la complejidad sea $O(x^n)$.

Analizando el problema anteriores se pueden obtener algunas mejoras. Parte de los caminos del arbol son iguales. Hay distintas ramas con nodos que tienen el mismo resto, y por lo tanto se puede calcular solo una vez. Este caso de resto igual en varios nodos, lo llamaremos subproblemas.



Subproblema: Calcular el óptimo(OPT) del cambio x debe usar el mínimo entre los subproblemas $X - C_j$ para $j = 1...n$.

Cada vez que paso por un subproblema se incremente en 1 para contar la cantidad de monedas a dar. Que seria: $1 + min\{subproblemas\}$.

Para la solución **recurrente**, podemos plantear:

$$\begin{cases} OPT(x) = 0 & si \quad x = 0 \\ \\ OPT(x) = 1 + min\{OPT(x - C_i)\} & si \quad x > 0 \end{cases}$$

El resultado con el minimo cambio sera OPT(x) y para poder calcularlo, necesito calcular los $x - 1$ óptimos anterios. Para evitar el recalculo, si calculo el optimo de algun resto, lo almaceno para no volver a calcularlo de nuevo. Ademas en cada subproblema debo analizar n comparaciones, lo cual impacta en la complejidad.

SOLUCIÓN ITERATIVA

```
1
2 OPT[0] = 0
3 Desde i=1 a X
4     minimo = +infinito
5     Desde j=1 a n
6         resto = i - C[j]
7         si resto > 0 y minimo > OPT[resto]
8             minimo = OPT[resto]
9
```

```
    OPT[i] = 1 + minimo

Retornar OPT[X]
```

Listing 8: Solución iterativa

La complejidad es $O(X * n)$ porque no solo depende de los diferentes tipos de monedas, tambien depende del parametro de entrada X. Se dice que es un algoritmo pseudo polinomial.

RECONSTRUIR LAS ELECCIONES

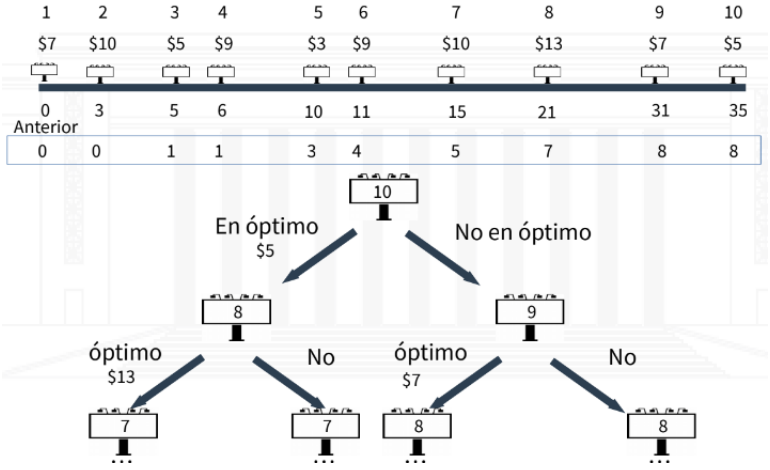
```

OPT[0] = 0
elegida[0] = 0
Desde i=1 a X
    minimo = +infinito
    elegida[i] = 0
    Desde j=1 a n
        resto = i - C[j]
        si resto > 0 y minimo > OPT[resto]
            elegida[i] = j
            minimo = OPT[resto]

    OPT[i] = 1 + minimo

resto = x
Mientras resto > 0
    Imprimir C[elegida[resto]]
    resto = resto - C[elegida[resto]]

Imprimir OPT[x]
```

Listing 9: Solución iterativa con reconstrucción

5.2 Problema de la Publicidad en la carretera

Sea una carretera de longitud M km, un conjunto de n carteles publicitarios en el intevalo $[0, M]$, cada cartel i tiene una posición x_i y un valor de ganancia r_i. Entonces queremos seleccionar carteles para maximizar la ganancia. Como restriccción ningún cartel puede estar a menos de 5 km de otro.

Podemos armar un arbol de decisión utilizando una funcion de *anteriores(i)*. La función anterior nos dice cual es el cartel anterior al i que cumple con la restriccción.



Para la solución **recurrente**, podemos plantear:

$$\begin{cases} OPT(i) = 0 & si \quad i = 0 \\ \\ OPT(i) = max\{r_i + OPT(anterior(i)), OPT(i-1)\} & si \quad i > 0 \end{cases}$$

El resultado con la máxima ganancia sera: $OPT(n)$.

SOLUCIÓN ITERATIVA

```
1
2  OPT[0] = 0
3  OPT[1] = r[1]
4
5  Desde i=2 a n
6
7      estaCartel = r[i] + OPT[anterior(i)]
8      noEstaCartel = OPT[i-1]
9
10     OPT[i] = max (estaCartel, noEstaCartel)
11
12 Retornar OPT[n]
```

Listing 10: Solución iterativa

SOLUCIÓN ITERATIVA - CARTELES SELECCIONADOS

```
1
2  OPT[0] = 0
3  OPT[1] = r[1]
4  elegidos[0] = false
5  elegidos[1] = true
6
7  Desde i=2 a n
8
9      estaCartel = r[i] + OPT[anterior(i)]
10     noEstaCartel = OPT[i-1]
11
12     Si estaCartel>noEstaCartel
13         elegido[i] = true
14     sino
15         elegido[i] = false
16
17     OPT[i] = max (estaCartel, noEstaCartel)
18
19 Retornar OPT[n]
```

Listing 11: Solución iterativa con reconstrucción

La complejidad temportal es $O(n)$ ya que solo hago sumas y comparaciones. La complejidad espacial es $O(n)$ porque se almacenan los n óptimos en un array.

SOLUCIÓN ITERATIVA - RECONSTRUIR

```
1
2  i = n
3
4  Mientras i>0
5      si elegido[i]
6          Imprimir i
7          i = anterior[i]
8      sino
9          i = i-1
10
11 Retornar OPT[n]
```

Listing 12: Solución iterativa

La complejidad temportal es $O(n)$. La complejidad espacial es $O(n)$.

CALCULO anterior de i

Se hace un apareo entre las posiciones del cartel x y el limite del mismo. El objetivo es armar un array de anteriores.

```
1
2  i=n
3  j=n-1
4
5  Mientras i>1
6      Si limite(n) >= posicion(j)
7          anterior[i] = j
8          i=i-1
9      sino
10         j=j-1
```

Listing 13: Solución iterativa

5.3 Programación de intervalos ponderados

5.4 Problema de Knapsack (mochila)

5.5 Problema de Subset Sum

Sea un conjunto de n elementos $E = \{e_1, e_2, ..., e_n\}$ donde cada elemento e_i cuenta con un peso asociado w_i.

Queremos seleccionar un subset de elementos de E con el mayor peso posible que no supere un valor W de peso máximo.

Para plantear una solución por **fuerza bruta**, un elemento puede estar o no. O sea que si tengo n elementos pueden existir 2^n combinaciones. Entonces la complejidad total esta acotado por $O(2^n)$.

5.6 Bellman Ford

Se extiendo el problema de hallar caminos minimos utilizando **aristas ponderas negativas**. Se puede hayar un camino global que pase por aristas ponderadas negativamente y que sea el optimo, en vez de utilizar un algotimo de reedy de *Dijkstra* que para este caso no seria óptimo.

Una solución por **fuerza bruta** seria, calcular para un grafo poderado **sin ciclos negativos**:

- Todos los costos de los caminos posibles de s a t de longitud 1.
- Todos los costos de los caminos posibles de s a t de longitud 2.
- ...
- Todos los costos de los caminos posibles de s a t de longitud n-1.

El camino mínimo tendra longitud n-1 como máximo sin ciclos negativos.

El algoritmo de **Bellman-Ford** halla el camino mínimo con aristas negativos utilizando programación dinámica.

ANÁLISIS

Para llegar desde "s" a un nodo n_i puede haber utilizado diferntes caminino y longitudes. Lo puede hacer a travez de sus nodos predecesores $pre[n_i]$.

Para poder llegar a n_i en j pasos, tengo que haber llegado a sus predeceroes en $j-1$ pasos. Asi sucesivamente hasta "s" se puede ir resolviendo *sub casos*.

Definimos $minPath(n, j)$ al camino mínimo hasta el nodo n_i con longitud máxima j.

SOLUCIÓN RECURRENTE

$$minPath('s', j) = 0$$
$$minPath(n_i, 0) = +\infty \qquad n_i \neq s$$
$$minPath(n_i, j) = min \begin{cases} minPath(n_i, j-1) \\ min\{minPath(n_x, j-1) + w(n_x, n_i)\} \end{cases} \qquad n_x \in pred(n_i)$$

- El camino mínimo a 's' para cualquier longitud es siempre 0.
- El camino mínimo a n_i al comienzo es infinito.
- TODO

SOLUCIÓN ITERATIVA

Definimos a $OPT[l][v]$ como el camino mínimo de "s" al nodo n con longitud$l$

El nodo "s" se encuentra en v=0 El nodo "t" se encuentra en v=n

```
Desde l=0 a n-1
    OPT[l][0] = 0
Desde v=0 a n-1
    OPT[0][v] = +infinito


Desde l=1 a n-1   // max longitud del camino
    Desde v=1 a n // nodo
        OPT[l][v] = OPT[l-1][v]
        Por cada p predecesor de v
            si OPT[l][v] > OPT[l-1][p] + w(p,v)
                OPT[l][v] = OPT[l-1][p] + w(p,v)

retornar OPT[n-1, n]
```

Listing 14: Algoritmo de requeridos con cupos

La complejidad del primer loop esta acotado por n. La segunda parte se ejecuta m veces por cada predecesor. O sea es $O(m * n)$

La complejidad espacial es m*n porque la matriz ocupa n*m

RECONSTRUIR LAS ELECCIONES

Agregar un nodo predecesor y almacenar en la posición i cual fue el predecesor del nodo.

¿Que pasa si hay un ciclo negativo?

Si en una iteración despues de haber llegado a la longitud maxima, cambia el minimo de al menos un nodo, entonces el grafo *tiene ciclos negativos*.

5.7 Problema de Maximo subarreglo

Se necesita calcular un subconjunto *contiguo de elementos* S tal que la suma de los valores sea la máxima posible.

El maximo subvector que termina en el elemento i, esta relacionado con el máximo subvector que termina en el elemento $i-1$.

SOLUCIÓN RECURRENTE

$$MAX(1) = v[1]$$
$$MAX(i) = max\{MAX(i-1), 0\} + v[i]$$

SOLUCIÓN ITERATIVA

```
1
2   MaximoGlobal = v[1]
3   MaximoLocal = v[1]
4   IdxFinMaximo = 1
5
6   Desde i=2 a n
7       MaximoLocal = max(MaximoLocal, 0) + v[i]
8
9       si MaximoLocal > MaximoGlobal
10          MaximoGlobal = MaximoLocal
11          IdxFinMaximo = i
12
13  Retornar MaximoGlobal
```

Listing 15: Solución iterativa

5.8 Problema de cuadrados minimos

Dado un conjunto de puntos $P = (x_1, y_1), (x_2, y_2), ..., (x_n, y_n)$, con $x_1 < x_2 < \cdots < x_n$. Usamos p_i para indicar un punto (x_i, y_i).

Queremos aproximimar mediante segmentos los puntos de P minimizando el error comentido. Los sementos se forman mediante *rectas de aproximación* hallando a y b. El calculo del error cometido se obtiene sumando las distancias de los puntos a las rectas.

Se agrega un parametro de penalización $C > 0$ por cada segmento que se agrega.

- A mayor "C" entonces: menos segmentos
- A menor "C" entonces: menos error

Al analizar una solución por **fuerza bruta** se obtiene una complejidad de $O(2^{n*n})$.

SOLUCIÓN RECURRENTE

Como no conocemos cual es el ultimo segmento, se elige el último segmento como aquel que **minimice el error general**. O sea que queremos minimizar el error del segmento, mas la constante c mas el error conocido en el *subproblema que contiene los puntos de segmentemos anteriores* sea el minimo entre todos los posibles.

$$OPT(i) = min_{1 \leq x \leq i}(e_{x,i} + C + OPT(x-1))$$
$$OPT(0) = 0$$

SOLUCIÓN ITERATIVA

```
1   OPT[0] = 0
2
3   Para todo para i,j con i <= j
4       Calcular e[i][j]
5
6   Desde j=1 a n
7       OPTIMO[j] = +infinito
```

```
8
9         Desde i=1 a n
10            segmento = e[i][j] + C + OPT[i-1]
11
12            si OPTIMO[j] > segmento
13                OPTIMO[j] = segmento
14
15    Retornar OPT[n]
```

Listing 16: Solución iterativa

Analizando la **complejidad temporal**, el calculo del optimo es $O(n)$, pero se calculan n óptimos, Por lo tanto esta partes es $O(n^2)$.

Pero como en la primer se itera sobre todos los pares posibles es $O(n)$. Y como el calculo del error es $O(n)$, la primer interación termina siendo $O(n^3)$, y este le gana a $O(n^2)$.

La complejidad total es $O(n^3)$.

Para el calculo de la **complejidad espacial**, los errores se almacenan en $O(n^2)$, mientras que los óptimos en $O(n)$. Por lo tanto la complejidad espacial total es de $O(n^2)$.

5.9 Problema del viajante

Sea un conjunto de n ciudades "C", un conjunto de rutas de costo de tránsito, existe una ruta que une cada par de ciudades.

Queremos obtener el circuito de menor costo que inicie y finalice en una ciudad y que pase por el resto de las ciudades *una y solo una* vez

Mediante **fuerza bruta** tenemos que calcular todos los ciclos posibles, y por lo tanto existen $(n-1)!$ ciclos de longitud $n-1$. Luego por cada ciclo calculamos su costo y nos quedamos con el mínimo. Por lo tanto la complejidad total es $O(n!)$.

Mediante el **algoritmo Belman-Held-Karp** lo resuelvo utilizando programación dinamica. Se puede decomponer como el mínimo entre los subproblemas menores con $(n-1)!$ hojas.

SOLUCIÓN RECURRENTE

Dado S un subconjunto de ciudades e i la ciudad donde estoy parado. **start** es la ciudad de partida. La siguiente es la ecución de recurrencia:

$$OPT(i, \{S\}) = min_{j \in \{S\}}(w(i,j) + OPT(j, \{S-j\}))$$
$$OPT(i, \emptyset) = w(i, start)$$

- El optimo i con el subconjuto s va a ser igual al minimo de los subproblemas que son elegir alguna de las ciudades que estan en s. Sumando el peso de i a j mas el optimo de partir de j hacia el resto de las ciudades (s-j).
- En el caso base, ya no quedan ciudades para visitas, entonces solo queda sumar el peso de ir de i a la ciudad de inicio *Start*.

SOLUCIÓN ITERATIVA Llamamos a C al conjunto de todas las ciudades, 1 es la ciudad inicial, y el resto de las ciudades estan numeradas de 2 a n.

```
1
2     Desde i=2 a n
3         OPT[i][0] = W[i][1]
```

```
4
5   Desde k=1 a n-2
6       Para todo subset S de C-{1} de tamanio k
7           Para cada elemento i de S
8               OPT[i, S-{i}] = +infinito
9
10              Por cada elemento j de S - {i}
11                  r=OPT[j, S-{i,j}] + w[j][i]
12
13                  si (r<OPT[i, S-{i}])
14                      OPT[i, S-{i}] = r
15
16
17  CamininoMinimo=+infinito
18  Desde j=2 a n
19      ciclo = OPT[i, S-{1, i}] + w[1, i]
20      Si (CamininoMinimo>ciclo)
21          CamininoMinimo = ciclo
22
23  Retornar CamininoMinimo
```

Listing 17: Solución iterativa

- La primer iteración se cargan los casos bases para las n ciudades.
- Despues desarrollamos los subproblemas, primero iteramos las ciudades que quedan por visitar
- Luego generamos las variantes de subset y por cada uno calculo el minimo y utilizo los subproblemas de tamaño menor, ver cual de todos es el minimo.

La complejidad total es $O(n^2 2^n)$

6 Redes de flujo

6.1 Conceptos

Se trata de problemas de flujos de trafico en redes. Por ejemplo, tubos de gas, autopistas, rutas de aviones, redes electricas.

Definiciones:

- Los **ejes** transportan algun tipo de flujo
- Los **vértices** actúan como conmutador de tráfico entre los diferentes ejes.
- Capacidad: cantidada máxima que un eje puede transportar.
- Fuente: Vértices que generan tráfico saliente.
- Sumidero:Vértice que absorbe tráfico entrante.
- Flujo: Cantidad transportada por eje.

Sea $G = (V, E)$ un grafo dirigido, para todo $e \in E$ llamamos $C_e \geq 0$ (valor entero) a su capacidad. Existe un único $s \in V$ llamado fuente (source). O sea no tiene ejes entrantes. Existe un único $t \in V$ llamado sumidero (sink). O sea no tiene ejes salientes. El resto de los vertices son internos como si fueran conmutadores de fuentes.

DEFINICION DE FLUJO El flujo $s - t$ es una funcion f que mapea cada e a un número real no negativo, $f : E \mapsto R^+$. Un flujo tiene las siguientes caracteristicas:

- (Condición de capacidad) Para cada $e \in E$, tenemos que $0 \leq f(e) \leq C_e$.
- (Condición de conversación) Para cada nodo v que no sean s y t , tenemos que:

$$\sum_{eintov} f(e) = \sum_{eoutofv} f(e)$$

PROBLEMA DE FLUJO MAXIMO

Definimos **corte de grafo** como:

Dividimos los nodos del grafo en dos conjuntos (A y B). Donde la fuente $s \in A$ y el sumidero $t \in B$. Cualquier flujo $s-t$ debe cruzar en algún punto de A a B. El corte define un limite al caudal máximo del flujo. Pero dos cortes diferentes, tienen capacidades de transporte maxima diferentes. Entonces deberia calcular todos los posibles cortes del grafo tomar el que maximice el flujo segun los limites de la red de transporte. Calcular de esta manera se torna inviable.

Entonces el problema de flujo maximo, será dada una red de flujo, encontrar el flujo de maximo valor posible.

6.2 Algoritmo Ford-Fulkerson

Calcula el maximo flujo a travez de una red.

Grafo residual

Dado un red de flujo G y un flujo f en G, **definimos el grafo residual G_f (de G con respecto a f)** a:

- Los mismos vértices de G,
- **Ejes hacia adelante**: Para cada $e = (u, v) \in E$ en el que $f(e) < C_e$. Lo incluimos en G_f con capacidad $C_e - f(e)$ [**capacidad residual** de flujo].
- **Ejes hacia atras**: Para casa $e = (u, v) \in E$ en el que $f(e) > 0$. Incluimos $e' = (v, u)$ con capacidad $f(e)$.

Cuello de botella

Sea P un **camino simple** $s - t$ en G_f, o sea que P no visita más de una vez el mismo vértice.

Difinimos **bottleneck(P,f)** a la capacidad residual mínima de cualquier eje de P con repecto al flujo f.

Lo máximo que se puede transportar es 20, para que no deje cumplir la condición de capacidad.

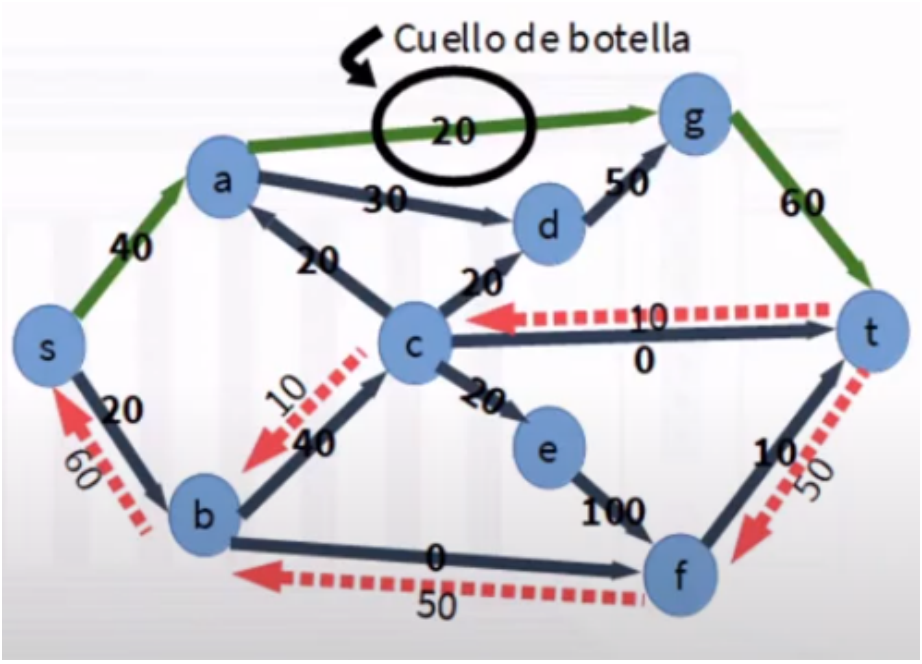
> *Con el grafo residual, podemos redireccionar el flujo en el camino original para aumentar el flujo total de la red.*

Tambien nos ayuda a saber cuanto flujo se esta trasportando por ese eje.

Llamamos P al camino de aumento (**augmenting path**):

- P es una caminio simple que va de s a t en G_f.
- P no visita el mismo nodo mas de una vez.

Ahora definimos la operación **augment(f,P)** el cual cede un nuevo flujo f' en eG



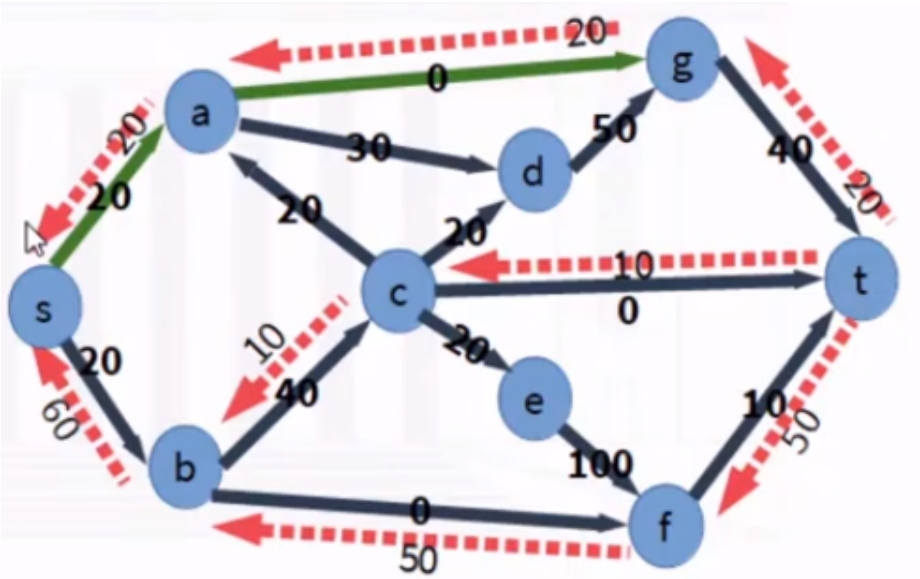

```
augment(f, p)
    Sea b = bottleneck(P, f)
    Para cada eje e=(u, v) perteneciente a P
        Si e=(u,v) eje hacia adelante
            f(e) += b en G
        sino si es eje para atras
            e' = (v,u)
            f(e') -=b en G
    Retornar f
```

Listing 18: Operación de augment

*¿Es valido el nuevo flujo?*PENDIENTE

Con el grafo residual y el camino de aumento definimos el *pseudocódigo de Ford-Fulkerson.*

```
Max-Flow
    Inicialmente  f(e)=0 para todo 'e' en G

    Mientras haya un camino s-t en Gf

        Sea P un caminio s-t simple en Gf
        f' = augment(f,P)

```




```
9           Actualizar f para ser f'
10          Actualizar Gf para ser Gf'
11
12
13      Retornar f
```

Listing 19: Operación de augment

La **complejidad** es $O(|E| * C)$ donde $|E|$ es la cantidad de ejes y C es la suma de todas las C_e de los ejes que salen de la fuente.

*¿Es óptimo?*PENDIENTE

> **El flujo retornado por el algoritmo Ford-Fulkerson es el flujo máximo**

Ademas podemos mediante BFS en G_f construir el corte mínimo s-t (A,B) obteniendo A y por diferencia B.

Consideraciones si las capacidades no son enteras:

- Si son racionales, multiplicar por minimo comun multiplo
- Si son irracionales, *no esta asegurado que el algoritmo termine.*

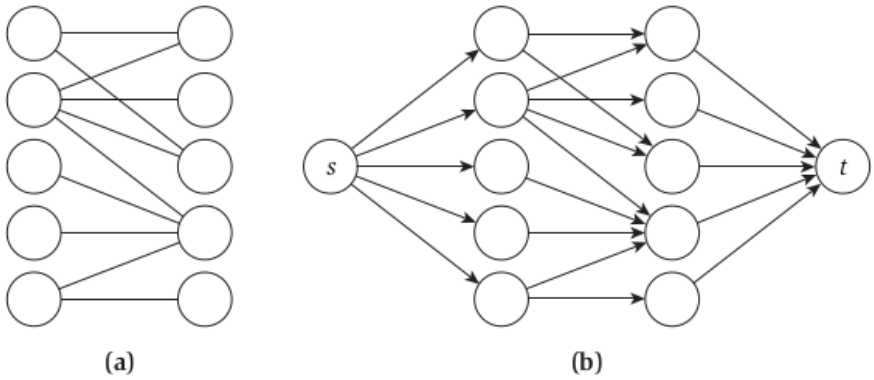
6.3 Variante: Circulación con demanda

Cada nodo pueder ser productor o consumir de flujo. O un nodo que no es consumidor ni productor de flujo.

6.4 Bipartite Matching Problem

Llamamos un grafo bipartito a $G = (V, E)$ un *grafo no dirigido*, puede particionarse en como $V = X \cup Y$, con la propiedad de que cada eje $e \in E$ se conecta en una punta con un nodo en X y la otra punta un nodo en Y. Un *matching* M en G es un subconjunto de ejes $M \subseteq E$ tal que cada nodo aparece en al menos un eje en M. Se necesita encontrar el set M de mayor tamaño posible. O sea la mayor cantidad de parejas.

Resolvemos el matching utilizando el problema de flujo máximo. Construimos una red de flujo G' como la siguiente imagen. Pasamos todos los ejes a ejes dirigidos de X a Y. Luego agregamos el nodo s y un eje (s, x) desde s a cada nodo en X. Tambien agregamos el nodo t y un eje (y, t) desde cada nodo en Y a t. Finalmente, le damos una capacidad de 1 a cada eje en G'



Resolvemos el problema de red de flujo máximo con G'. Obtenemos el flujo máximo $s - t$. Entonces *El valor del flujo total es igual al tamaño del matiching máximo.*

Analisis Pendiente

6.5 Diseño de encuentas

Considere el problema de una compañia que vende k productos y que tiene una base de datos con el historias de las compras de todos sus clientes. La compañia desea enviar encuestas con preguntas personalizadas a un grupo particular de n clientes, para determinar que productos la gente prefiere sobre el total.

Lineamientos para la encuesta:

- Cada cliente recibira preguntas acerca de cierto subconjunto de productos.
- Un cliente solo puede contestar sobre los productos que él o ella haya comprado.
- Cada cliente sera preguntado sobre un número de productos entre c_i yc_i'
- Cada producto debe tener entre p_j y p_j' preguntas de clientes distintos.

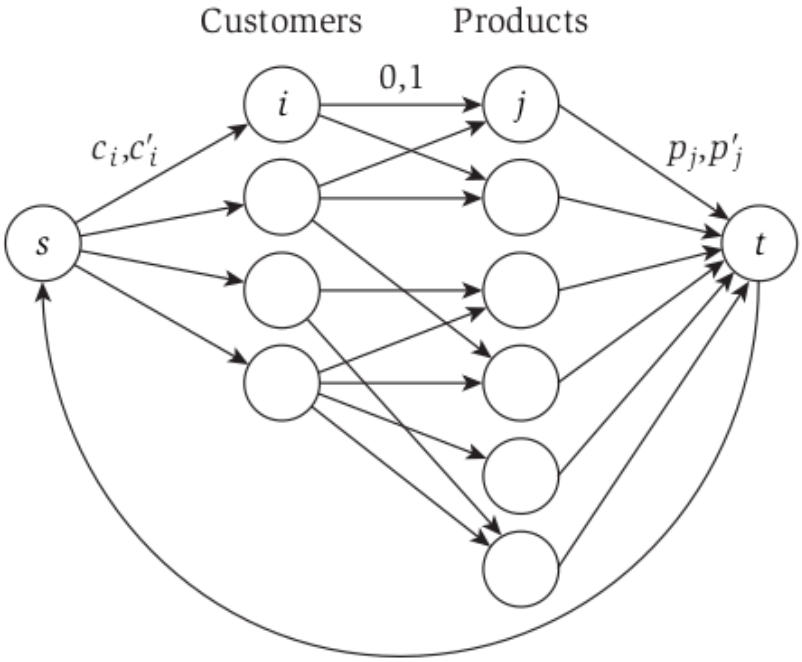
El problema de diseño de encuentas toma como input un *grafo bipartito* G cuyos nodos son clientes y productos, y hay un eje entre un cliente i y un producto j si el cliente compro el producto j. Mas aún, por cada cliente $i = 1, 2, ..., n$ tenemos la limitante de $c_i \leq c_i'$ en el numero de productos en el que un cliente puede constestar; por cada producto $j = 1, ..., k$, tenemos la limitante $p_j \leq p_j'$ en el número de cliente distintos que se pueden consultar por cada producto.

El problema se resuelve reduciendo este a un problema de red de flujo en G' con demanda y un limite inferior.

Para obtener un grafo G' de G, necesitamos:

- Orientar los ejes de G desde los clientes a los productos.
- Agregar un nodo ficticio s con los ejes (s, i) por cada cliente $i = 1, ..., n$.
- Agragar un nodo ficticio t con los ejes (j, t) por cada producto $j = 1, ..., k$.

La circulacion en la red, corresponde con la manera en la que se tienen que realizar las preguntas.



Se debe pasar de un problema de circulación con *demanda y limite inferior* a un problema de circulación con demanda y luego a un problema de flujo máximo. Finalmente se resuelve con Ford-Fulkerson.

Una vez obtenido el flujo máximo:

- El flujo que va de (t, s) corresponde al número total de preguntas a realizar.
- El flujo en los ejes (s, i) es el número de productos que deben contener el cuestionario para cada cliente i.
- El flujo en los ejes (j, t) corresponde con él numero de clientes que deben ser preguntados para el producto j.
- Por ultimo, aquellos ejes (i, j) con flujo 1, corresponden a preguntar al cliente i sobre el producto j.

6.6 Problema de Selección de proyectos

Dado un conjunto P de proyectos para seleccionar y cada proyecto $i \in P$ tiene asociado una ganancia p_i, el cual puede ser *positivo* como *negativo*. Algunos proyectos son requisitos de otros proyectos, y modelaremos esta relación mediante un *grafo dirigido sin ciclos* $G = (P, E)$. Los nodos de G son los proyectos y hay un eje (i, j) para indicar que un proyecto i puede ser seleccionado solo si el proyecto j es tambien seleccionado. Un proyecto i pude tener muchos prerequisitos, y puede haber muchos proyectos j que pueden ser parte de esos prerequisitos. Un conjunto de proyecto de $A \subseteq P$ es *viable* si los prerequisitos de cada proyecto de A tambien pertenecen a A:

Por cada $i \in A$ y cada eje $(i, j) \in E$, tenemos que $j \in A$

Estos prerequisitos vendrian a ser las *restricciones de precedencia*. La ganancia del conjunto de proyectos se define como:

$$profit(A) = \sum_{i \in A} p_i$$

El *problema de selección de proyectos* seleccionar el conjunto de proyectos viables con la maxima ganancia.

7 Problemas NP

7.1 Clasificación

7.1.1 Clase P

La clase P consiste en aquellos problemas que pueden resolverse en tiempo polinomial (eficientemente).

Un algoritmo A resuelve **eficientemente** un problema S si para toda instancia I de S, encuentra la solución en tiempo polinomico, entonces existe una constante $k/A = O(n^k)$ y donde n es el tamaño de la entrada del problema. Ejemplo, Gale Shapley resuelve el problema de "Stable Marriage Problem" en $O(n^2)$

Se conoce como P al conjunto de problemas de decisión para los que existe un algoritmo que lo resuelva en *forma eficiente*.

Un algoritmo B **certifica eficientemente** un problema de decisión S si para toda instancia I de S, dado un certificado t que contiene evidencia de la solución $s(i)$ es *"Si"*, entonces existe una constante $k/B = O(n^k)$.

O sea que el algoritmo B va a recibir dos parametros, la instancia I y el certificado T. Responde si o no. Por ejemplo, en el problema de la moneda, seria las cantidades de monedas a dar, y certificado seria la solución conocida. Para validar, se ejecuta el algoritmo de certificación con el certificado T.

7.1.2 Clase NP

Se conoce como NP al conjunto de problemas de decisión para los que existe un algoritmo que lo verifique (certifique) en tiempo polinomial (eficientemente).

¿$P \subseteq NP$?. Si el problema $Q \in P$, existe un algoritmo $A = O(n^k)$ que lo resuelve. Y podemos definir como:

```
1
2   B(I, t)
3       s = A(I)
4       Si s == t
5           retornar "si"
```

```
6         retornar "no"
```

Listing 20: Algoritmo B

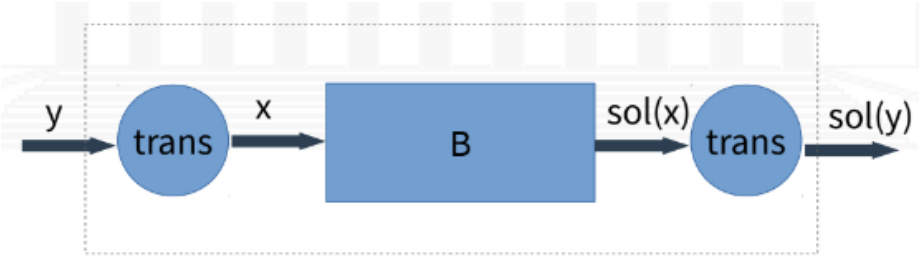
Que certifica el problema Q y lo hace en tiempo polinomial. Entonce se cumple:

$$Q \in P \implies Q \in NP$$

¿$NP \subseteq P$?. Es un problema sin resolver.

7.2 Reducciones

Reducir un problema a otro conocido.



Una reducción polinomial corresponde a una reducción en la que ambas transformaciones se realizan en tiempo polinonimial.

Sean X, Y problemas, diremos $Y \leq_p X$, se lee Y es polinomialmente reducible (en tiempo) a "X"

Si podemos transformar cualquier instancia de Y en una instancia de X en tiempo polinomico (tractable).

Para **comparar problemas** con reducciones, sean X, Y problemas, si $Y \leq_p X$, diremos que el problema X es al menos tan dificil que el problema Y

Para **acotar un problema** a la clase P.

- Sean X, Y problemas si $X \in P$ y $Y \leq_p X$ entonces $Y \in P$, porque X es

igual de complicado que Y. Ejemplo:

$$MAXMATCHING \leq_p MAXFLOW$$
$$MAXFLOW \in P \implies MAXMATCHING \in "P"$$

- Sean X, Y problemas si $Y \notin P$ y $Y \leq_p X$ entonces $X \notin P$, porque X es igual de complicado que Y.

Las siguientes son propiedades de reducciones:

- *Equivalencia*: Sean X, Y problemas si $Y \leq_p X$ y $X \leq_p Y$ entonces X e Y tiene la misma complejidad.
- *Transitividad*: Si $Z \leq_p Y$ y $Y \leq_p X$ entonces $Z \leq_p X$

7.3 Clase NP completo

7.3.1 Problema de satisfabilidad booleana - SAT

Dado un conjunto de variable booleanas que definen una expresión booleana, determinar si existe una asignación de valores de las variables, tal que el resultado de la expresión es "TRUE".

Sea una instancia I del problema **SAT** $\in NP$ y un certificado que corresponde a un valor de asignación de cada variable.

Podemos certificar en tiempo polinomial si esa asignación de variables producen un resultado "TRUE".

El **teorema de Cook-Levin** dice, sea $X \in NP$ entonces $X \leq_p$ Boolean satisfability problem (SAT). O sea, **que todo problema perteneciente a NP es a lo sumo tan complejo de resolver que SAT**.

NP-HARD: Sea un problema X tal que para todo problema $Y \in NP$ y $Y \leq_p X$, entonces $X \in NP - HARD$. **X es al menos igual de dificil que cualquier problema NP**.

NP-Complete: Sea un problema X tal que para todo problema $X \in$NP-HARD y $Y \in$ NP, entonces $X \in NP - C$. **X es uno de los problemas mas dificiles dentro de NP**.

Ejemplo de uso, para cada problema de $X \in NP$ que analiza Richard Carp, toma un problema demostrado como NP-C y lo reduce a X, con esto X pasa a ser un problema **NP-C**.

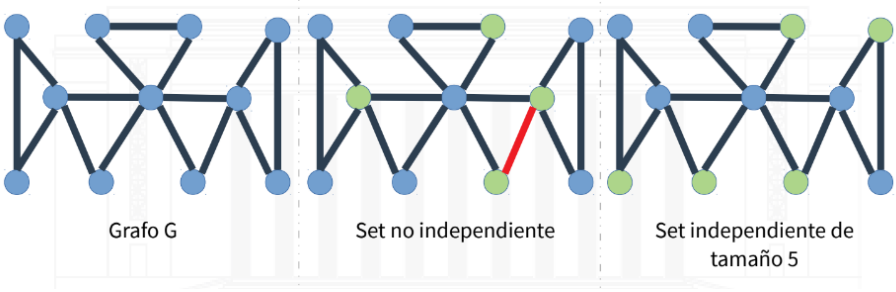
Probar que un problema es *NP-C*. Sea el problema X de decisión. Probamos:

- Probar que $X \in NP$, definiendo un certificado eficiente:
- Probar que $X \in NP$-HARD, dado un problema $Y \in NP$-C, reducir polinomialmente Y a X. Eligiendo el problema Y correcto para reducir, podemos obtener $Y \leq_p X$ y agregar a X en la clasificación de los problemas *NP-C*.

7.4 Problema de conjunto independiente

Sea un grafo $G = (V, E)$, un valor K, determinar si existe un conjunto independiente de nodos de como mucho tamaño k.

Definimos un conjunto de nodos $C \subseteq V$ es independiente si no existe $a, b \in C$ tal que existe eje $(a, b) \in E$ y el **tamaño del conjunto independiente** corresponde con la cantidad de nodos dentro del conjunto C.



Dado un grafo $G = (V, E)$ con tamaño k de conjunto y un certificado T igual al subconjunto de nodos. Si se puede verificar en tiempo polinomial con $|T| = K$ que:

$$\forall a, b \in T, !\exists (a, b) \in E \implies INDEPENDENTSET \in NP$$

7.4.1 3-SAT

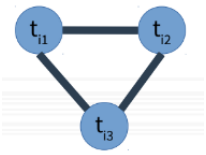
Es una variante de SAT donde cualquier instancia de SAT se puede reducir polinomialmente a 3SAT:

$$SAT \leq_p 3SAT \implies 3SAT \in NPComplete$$

Dado $X = x_1, ..., x_n$ conjunto de n variables booleanas $= \{0, 1\}$ y k *clausulas* booleanas $T_i = (t_{i1} \vee t_{i2} \vee t_{i3})$ con cada $t_{ij} \in X \cup \overline{X} \cup 1$. Entonces debemos **determinar si existe asignación de variables** tal que $T_1 \wedge T_2 \wedge ... \wedge T_k = 1$

7.4.2 Reducción de 3-SAT a INDEPENDENT-SET

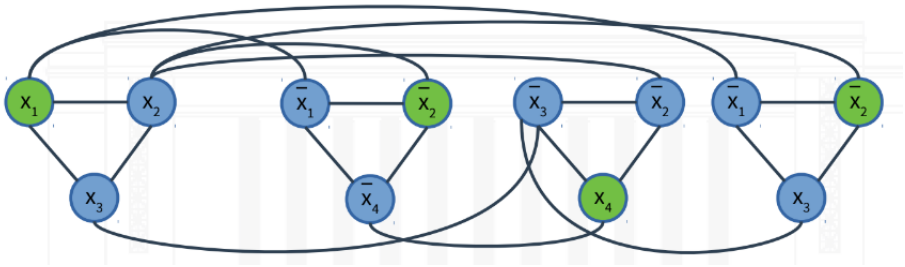
Por cada *clausula* $T_i = (t_{i1} \vee t_{i2} \vee t_{i3})$ **vamos a generar tres vertices entre si**. Y por cada $t_{ij} = x_a, t_{kl} = \overline{x_a}$, crear un eje entre t_{ij} y t_{kl}.



El grafo resultante G corresponde a una instancia del problema *INDEPENDENT-SET* con k=numeros de clausulas en la expresión. Ejemplo, sea la expresión:

$$E = (x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_4}) \wedge (\overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3)$$

Reducimos polinomialmente y resuelvo:



Entonces, luego resuelvo 3SAT con $x_1 = 1$, $\overline{x_2} = 1 \implies x_2 = 0$, $x_4 = 1$ y elijo $x_3 = 0$ porque en este caso es indistinto.

Por lo tanto, como INDEPENDENT-SET $\in NP$ y $3SAT \leq_p INDEPENDENTSET$ NP.

Entonces INDEPENDENT-SET $\in NPComplete$

7.5 Problema de cobertura de vertices

Sea un grafo $G = (V, E)$ diremos que existe un conjunto $S \subseteq V$ es una cobertura de vétices si:

$$\forall\ eje\ e \in E = (u, v), u \in S\ \ y/o\ \ v \in S$$

Donde u y/ó v pertenecen al conjunto S.

El **problema de decisión** sera que dado un grafo $G = (V, E)$, determinar si existe una conbertura de vértices (VERTEX-COVER) de tamaño al menos k. **El problema de optimización busca el subconjunto de menor tamaño**.

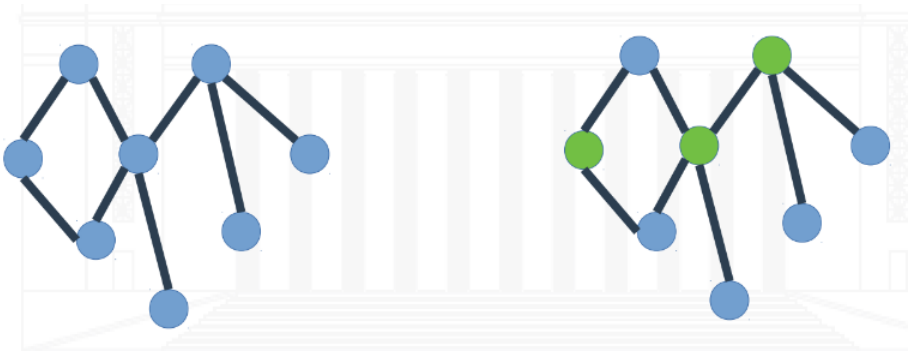


Figure 1: *Ejemplo de conbertura de vertices de k=3.*

Para ver si VERTEX-COVER pertenece a un **problema NP**, verificamos de la siguiente manera:

Sea un grafo $G = (V, E)$ y un certificado T como un conjunto de nodos de V que forman el cubrimiento. Verificamos que:

$$\forall\ eje\ e \in E = (u, v),\ si\ u \in T\ \ o\ \ v \in T\ \implies\ O(V * E)$$

observamos si uno de estos vertices pertenece al certificado y lo podemos ver en tiempo polinomial de $O(V * E)$ y si ademas $|T| = k$, entonces el certificado nos da la respuesta con tamaño k, y por lo tanto **VERTEX-COVER** $\in$ **"NP"**

Para ver si VERTEX-COVER pertenece a un **problema NP-Completo**, utilizaremos INDEPENDENT-SET.

7.6 Problema de cobertura de conjunto

Sea un conjunto U de n elementos. Una colección $S_1, ..., S_m$ de subconjuntos de U. Problema de decisión: ¿Existe una colección de como mucho k de los subconjuntos cuya unión es igual a U?

La idea es probar que SET-COVER es NP-COMPLETO. Sea U un conjunto de elementos, K tamaño buscado, los subset $S_1, ..., Sm$ y un T certificado con los indices de los subconjuntos del conjunto.

Verificamos que $|T| = k$ para todo elemento en U, si existen en algunos de los subconjuntos de T. Por lo tanto se puede hace en tiempo polinomial y se puede afirmar que **SET-COVER es "NP"**.

Luego elegimos un problema que previamente se haya demostrado que es NP-Completo. Para esto vamos a usar VERTEX-COVER. Vamos a intentar demostrar que VERTEX-COVER $\leq_p$ SET-COVER. El algoritmo de reducción sera:

Para la **reducción de VERTEX-COVER a SET-COVER**, partimos de $G = (V, E)$ y k. Queremos que todos los ejes queden cubiertos y construimos un conjunto de elemtentos U=E. Por cada vertice $v \in V$, creamos un subconjunto S_v con todos los ejes incidentes a el, y mantenemos en k la cantidad de subconjuntos a buscar para cubrir U. Toda esta transformación se puede hacer en tiempo polinomial.

Si encontramos el subconjunto, eso nos dira que vertices seleccionar.

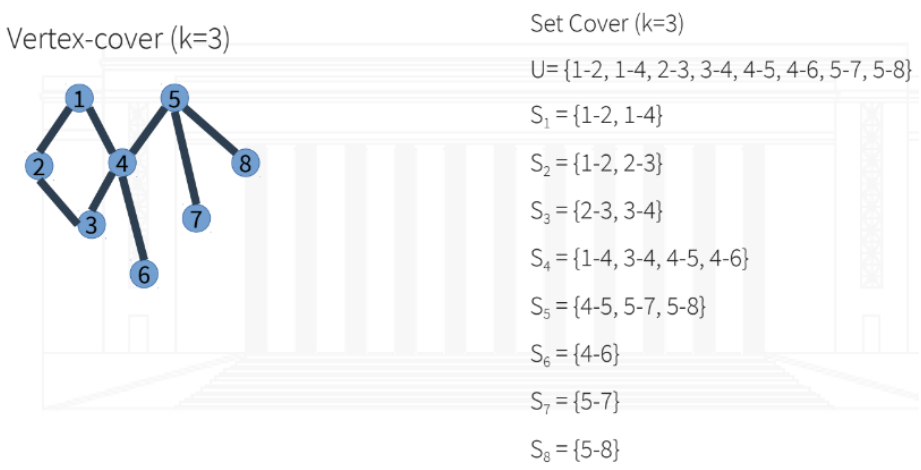


Figure 2: *Ejemplo de reducción de VERTEX-COVER a SET-COVER.*

Si se resuelve SET-COVER, se obtiene los subconjuntos que corresponden a los nodos resultantes en el problema de VERTEX-COVER.

Resumiendo, demostramos que SET-COVER es NP-COMPLETO, porque reducimos VERTEX-COVER a SET-COVER en tiempo polinomial, y si resolvemos cualquier instancia de set-cover, podemos resolver cualquier instancia de vertex-cover.

7.7 Problema 3 Dimensional Matching

Dado 3 conjuntos disjuntos X, Y, Z de tamaño n cada uno y un conjunto $C \subset X \times Y \times Z$ de tripX, Y, Zlas ordenadas. Determinar si existe un subset de n triplas ene C tal que cada elemento de $X \cup Y \cup Z$ sea contenido exactamente en una de esas triplas.

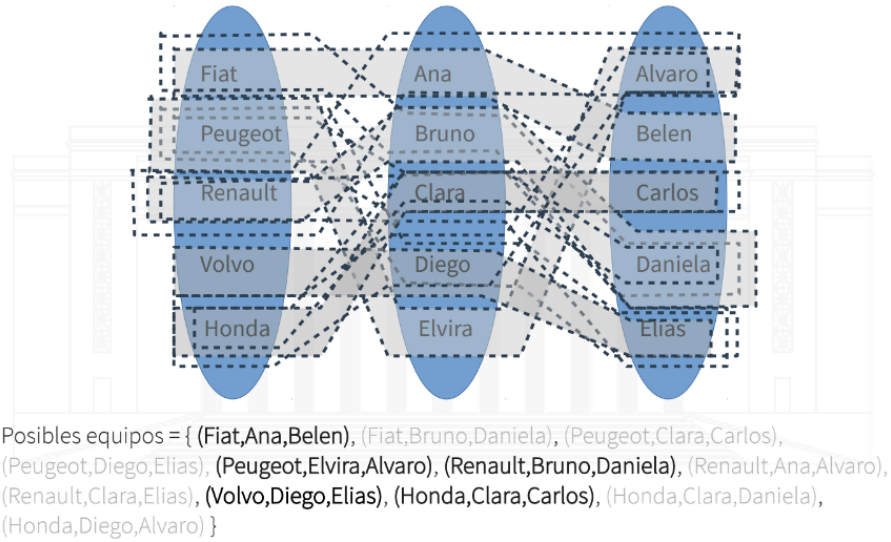


Figure 3: *Ejemplo de uso de 3DM.*

7.7.1 3DM pertenece a los problemas NP?

Para verificar que 3DM pertenece a NP, dado 3 conjuntos disjuntos X, Y, Z, $C = (x, y, z)$ conjunto de triplas y un certificado T, de triplas con un subconjunto de C.

Podemos certificar en tiempo polinomial que todo elemento en X, Y y Z, se encuentra 1 y solo 1 vez en algun T_i. Y si $|T| = n$, entonces:

$$3DM \in NP$$

7.7.2 3DM pertenece a los problemas NP-HARD ?

Ahora queremos probar que 3DM pertenece a los problemas NP-HARD. Probamos que $3SAT \leq_p 3DM$.

Sea I instancia de *problema 3SAT* con n variables $x_1, ..., x_n$ y k clausulas $c_1, ..., c_k$, reducimos en tiempo polinomial la instancia I a un *problema 3DM.*

> **Gadget:** Formas o plantillas pre-armadas para hacer reducción de cualquier instancia. Se van a utilizar gadget para las variables y para las clausulas.

7.7.3 Reducción de 3SAT a 3DM

Por cada variable x_i creamos un gadget formado por los siguientes conjuntos:

1. $A_i = \{a_{i,1}, a_{i,2}, ..., a_{i,2k}\}$, seria el nucleo del gadget (2k elementos)
2. $B_i = \{b_{i,1}, b_{i,2}, ..., b_{i,2k}\}$, seria las puntas del gadget (2k elementos)

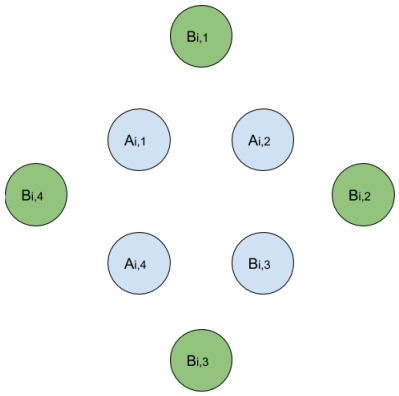


Figure 4: *Ejemplo para 2k elementos. Para la variable xi y k=2 clausulas, se generan 4 elementos por conjunto*

Por cada variable x_i creamos triplas que van a estar formados por dos elementos del nucleo del gadget y 1 elemento de la punta del gadget:

$$t_{ij} = \{a_{i,j}, a_{i,j+1}, b_{i,j}\}$$

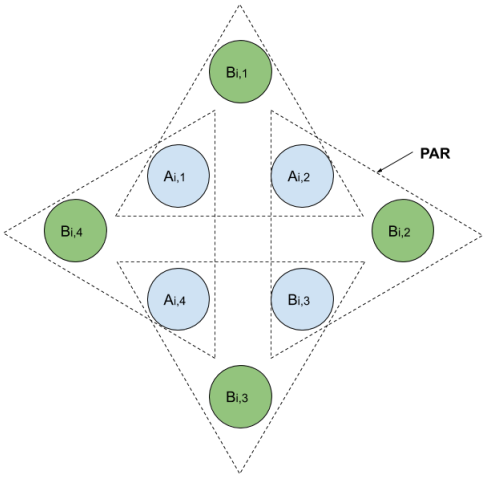


Figure 5: *Las triplas se superponen. Cada uno de los elemento del nucleo va a formar parte de dos triplas. Cada elemento de la punta estara unicamente dentro de una tripla. Luego numeramos las triplas segun el valor de j y lo hacemos en el orden de las agujas del reloj, llamamos tripla par, si j es par y tripla impar si j es impar.*

Por cada clausula creamos un set de elementos llamados nucleos:$C_j = \{p_j, p'_j\}$. Por cada variable i en la clausula C_j:

- Si contiene la variable $\overline{x_i} \rightarrow$ creamos la tripla $p_j, p'_j, b_{i,2j-1}$
- Si contiene la variable $x_i \rightarrow$ creamos la tripla $p_j, p'_j, b_{i,2j}$

CONTINUACION PENDIENTE.

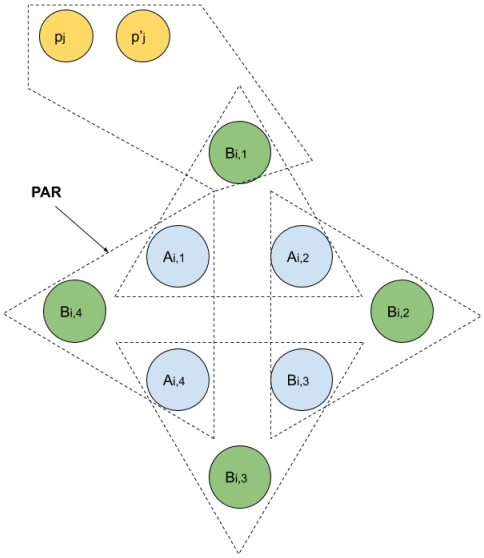


Figure 6: *Ejemplo con clausula 1: con la variable $\overline{x_i}$=1 y j=1, la tripla formada sera* $p_1, p'_1, b_{1,1}$.

7.8 Ciclo Hamiltoneano

Sea un grafo G=(V,E) dirigido definimos un ciclo C en G como hamiltoneano si visita cada vértice 1 y solo 1 vez. Es un recorrido que inicia y finaliza en el mismo vertices.

El problema de decisión sera HAM-CYCLE, sea G=(V,E) grafo dirigido, ¿existe un ciclo hamiltoneano?

Para saber si el HAM-CYCLE pertenece a NP, dado un grafo G=(V,E) y un certificado $T = \{t_0, ..., t_{|v|}\}$ lista ordena de vertices. Puedo verificar en tiempo polinomial los siguientes puntos:

1. La cantidad nodos en T es igual a la cantidad de vertices en V. $|T| = |V|$ y $t_0 = t_|M|$.
2. Todos los vértices de V estan en T.
3. Para todo par de vertices $t_i, t_{i+1} \in T$ existe un eje direccionado $(t_i, t_{i+1}) \in E$ que los une. O sea seguimos el camino ordenado de nodos.

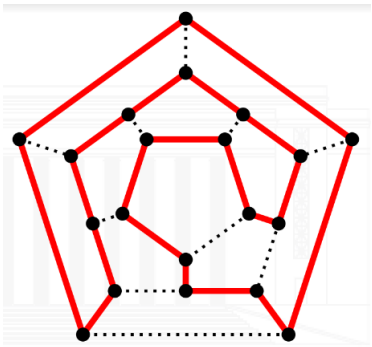


Figure 7: *Ejemplo de un ciclo de hamilton.*

Por lo tanto podemos afirmar que **HAM-CYCLE**$\in NP$.

NP C - PENDIENTE

7.9 Problema del caballo

Se conoce como *problema del caballo*, a encontrar una serie de movimientos del caballo de ajedrez que partiendo de una posición del tablero recorra todos los casilleros y regrese a la casilla inicial sin pisar dos veces la misma casilla.

¿Es posible que un caballo de ajedres desde una posición determinada recorra todo el tablero sin pisar dos veces la misma casilla?

Para tableros nxn en "Solution of the knight's Hamiltonian path problem on chessboards" Axel Conrad probaron que:

- si $n \geq 5$ se puede hallar un *camino* hamiltoneano.
- si $n \geq 6$ se puede hallar un *ciclo* hamiltoneano.

Es un caso particular del ciclo hamiltoneano y que se puede resolver en forma polinomial para ambos casos.

> En particular no interesa analizar si el problema es un NP-Completo el cual no podemos resolver ó existe un caso particular como este que si.

7.10 Problema del viajante de comercio

Un viajante debe recorrer n ciudades $v_1, v_2, ..., v_n$ partiendo de v_1 se debe construir un tour visitando cada ciudad una vez y retornar a la ciudad inicial.

Caminos: Para todo par de ciudades:

- Se especifica una distancia $d(v_x, v_y)$ entre las ciudades.
- No necesariamente hay simetria: $d(v_x, v_y)$ puede ser diferente a $d(v_y, v_x)$
- No necesariamente se cumple la desigualdad triangular: $d(v_i, v_j)+d(v_j, v_k) > d(v_i, v_k)$

Esto es un caso generalizado.

Problema de decisión: Dado n ciudades y las distancias entre cada par de ciudades, determinar un tour(o ciclo) de distancia total menor a k. k sera un parametro del problema.

7.10.1 Problema del viajante es NP?

Sea n ciudades, las distancias entre cada par de ciudades, T certificado = tour de ciudades y k distancia como limite. Se debe verificar T contiene todas las ciudades (solo 1 vez) y termina y comienza en la misma. Y por ultimo, la suma de las distancias recorrida es menor a k. Todas estas verificaciones se puede hacer en timpo polinomial y por lo tanto el problema del viajante es NP.

7.10.2 Problema del viajante es NP Completo?

Utilizamos HAM-CYCLE para reducir al problema del viajante. Sea una instancia I de HAM-CYCLE $G = (V, E)$.

1. Por cada vértice $v_i \in V$, creamos una ciudad v'_i
2. Por cada arista $e_{i,j} \in E$, definimos la distancia $d(v'_i, v'_j) = 1$
3. Aquellas distancias que no estan definidas(Las que en el grafo original no tienen aristas) las creamos con valor 2.

Tambien definimos el valor K, el cual es la distancia limite que tienen que medir el camino. Ponemos como valor $k = |V|$ (numero de vertices del grafo original)

Solucionamos el viajante con k definido si existe un camino con longitud k, entonces existe ciclo hamiltoneano. Si se encontro un ciclo cuya longitud es mayor a k, es porque se tomo un camino con distancia 2, los cuales no existen en el grafo original, y entonces no puede formarse un ciclo hamiltoneano en el grafo original.

Si encontramos el ciclo en le viajante, cuya longitud sea igual a k, y como cada ciudad corresponde a un vertice del grafo original, se puede tranformar a un ciclo hamiltoneado. Esta transformación es polinomial y por lo tanto cualquier problema hamiltoneano lo podemos reducir a un problema del viajante. Como antes demostramos que el problema del viajante es NP, entonces podemos decir que el problema del viajante es NP-Completo.

VIAJANTE $\in NP$ y $HAM - CYCLE \leq_p VIAJANTE \implies$
VIAJANTE $\in NPComplete$

7.11 Coloreo de Grafos

8 Algoritmos Randomizados

Un algoritmo randomizado es aquel que resuelve un problema P utilizando como parametro extra una cadena aleatoria "r". Las decisiones de ejecución se realizan teniendo en cuenta la lectura de la cedena aleatoria.

Clase de complejidad RP: Se conoce como "RP" o "R" a aquellos problemas de decisión para los que existe un programa "M" randomizado que se ejecuta en tiempo polinomial. Y que para toda instancia I del problema:

- Si la instancia I debe tener como resultado "Si", entonces:

$$Probabilidad(M(I,r) = "Si") \geq 1/2$$

- Si la instancia I debe tener como resultado "No" (no debe tener falsos positivos), entonces:

$$Probabilidad(M(I,r) = "No") = 0$$

Clase de complejidad co-RP: Se conoce como "RP" o "R" a aquellos problemas de decisión para los que existe un programa "M" randomizado que se ejecuta en tiempo polinomial. Y que para toda instancia I del problema:

- Si la instancia I debe tener como resultado "Si" (no debe tener falsos negativos), entonces:

$$Probabilidad(M(I,r) = "No") = 0$$

- Si la instancia I debe tener como resultado "No", entonces:

$$Probabilidad(M(I,r) = "No") \geq 1/2$$

Clase de complejidad ZPP: Se conoce como *zero-error probabilistic P* (ZPP) a aquellos problemas de decisión que pertenecen a "RP" y "co-RP". Implica que para toda instancia I del problema podemos ejecutar el algoritmo en RP y co-RP, entonces en tiempo polinomial tendremos 3 respuestas posibles: "Si", "No" y "No es posible".

La repetición de un número no determinado de ejecuciones nos asegura obtener el resultado correcto. Este tipo de clase de complejidad corresponde a los algoritmos de **Las Vegas**, en la intersección RP $\cap$ co-CP.

Clase de complejidad BPP: Se conoce como *bounded-error probabilistic P* (BPP) a aquellos problemas de decisión para los que existe un programa "M" randomizado que se ejecuta en tiempo polinomial. Y que para toda instancia I del problema:

- Si la instancia I debe tener como resultado "Si", entonces:

$$Probabilidad(M(I,r) = "Si") \geq 2/3$$

- Si la instancia I debe tener como resultado "No", entonces:

$$Probabilidad(M(I,r) = "Si") \leq 1/3$$

No podemos estar seguros. Si el resultado es correcto, podemos afirmarlo con "alta probabilidad". Este tipo de clase de complejidad corresponde a los algoritmos de **Monte Carlo**.

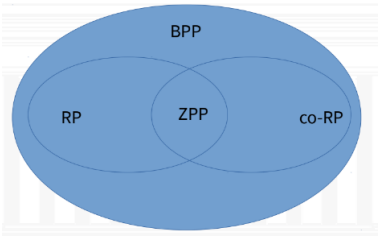


Figure 8: *Relación entre clases.*

8.1 Mezcla aleatoria

Sea un conjunto A de n elementos que queremos generar un listado de A ordenado aleatoriamente. Mezclar conjuntos se utilizan en *juegos de azar, reproducción de musica aleatoria, modelos estadisticos simulaciones, pruebas de complejidad algoritmica y otros mas.*

Permutación por ordenamientos: Para cada i elemento en A, generaremos un numero p_i aleatorio como su clave. Utilizando p_i para cada elemento a_i como "clave", ordenaremos "A". Podemos elegir cualquier metodo de ordenamiento como caja negra para resolverlo. O sea ordenamos basandonos en la clave aleatoria que generamos para cada elemento.

```
1 Sea A[1..n] conjunto a ordenar
2 Sea P[1..n] vector numerico // vector de prioridades.
3
4 Desde j=1...n
5     P[j] = random_value(1...x)
6
7 Ordenamos A utilizando P como clave
8
9 Retornar A
```

Listing 21: Algoritmo de permutación por odenamiento

Este algoritmo tiene un problema y es que si se generan claves repetidas, en metodos de ordenamientos como los estables, no se realizara la permutación y de esta forma no se podra obtener una permutación aleatoria uniforme (en terminos de probabilidad).

Para disminuir la posibilidad de claves repetidas, tenemos que tomar las siguientes acciones:

1. Podemos establecer un valor de X muy alto. $X >>> n$ (Por ejemplo n^5).
2. Se puede agregar un registro de claves utilizadas y volver a seleccionar otra clave si surge una ya utilizada. En el peor de los casos, se puede obtener siempre la misma clave y por consiguiente entrar en un loop infinito, si se elige mal el valor de x.

La complejidad temporal es $O(nlog(n))$ por que la generación de claves es $O(n)$ y el ordenamiento es $O(nlog(n))$ por lo que mandan el ordenamiento. La

complejidad espacial es $O(n)$.

Según el análisis de uniformidad cualquier permutación tiene probabilidad $1/n!$. Por lo tanto este método genera una *permutación aleatoria uniforme*.

Algoritmo de mezcla de Fisher-yates: Tambien se conoce como "barajado de sombrero". Se introducen todos los números en un sombrero, se agita el contenido(se mezclan) y se van sacando de a uno y se listan el mismo orden en el que se sacan hasta que no queden ninguno.

La descripción algoritmica es, para cada elemento A[i], generamos un calor x al azar entre i y n. Luego intercambiamos A[i] con A[x].

```
Sea A[1..n] conjunto a ordenar

Desde i=1...n
    intercambiar A[i] con A[random_value(i...n)]

Retornar A
```

Listing 22: Algoritmo de Fisher-yates

Según el análisis de uniformidad cualquier permutación tiene probabilidad $1/n!$. Por lo tanto este método genera una *permutación aleatoria uniforme*.

La complejidad temporal es O(n) porque tenemos n interaciones, y dentro de cada interación tenemos un intercambio que es O(1) y un ramdom que tambien es O(1). La complejidad espacial es O(1) porque todo el algoritmo se puede hacer sobre el mismo vector.

Ejemplo con $A = \{a_1, a_2, a_3, a_4\}$:

i	Rand	Intercambio
1	[1 a 4] → 2	$\{a_1,a_2,a_3,a_4\} \rightarrow \{a_2,a_1,a_3,a_4\}$
2	[2 a 4] → 3	$\{a_2,a_1,a_3,a_4\} \rightarrow \{a_2,a_3,a_1,a_4\}$
3	[3 a 4] → 3	$\{a_2,a_3,a_1,a_4\} \rightarrow \{a_2,a_3,a_1,a_4\}$
4	-	$\{a_2,a_3,a_1,a_4\}$

Figure 9: *Ejemplo de mezcla de Fisher-yates*

8.2 Problema de K conectividad de en un grafo

Sea $G = (V, E)$ grafo conexo y no dirigido, deseamos saber ¿cuantos ejes se pueden remover antes que G deje de ser conexo?. Se espera encontrar la minima cantidad de ejes para que se un grafo conexo.

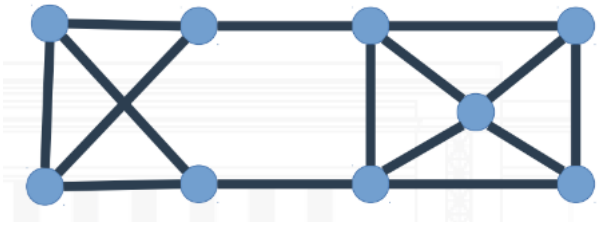


Figure 10: *Ejemplo de k conectividad*

Analizamos el problema y lo podemos pensar como encontrar el corte global mínimo del grafo si analizamos toda posible subdivisión A-B del grafo en 2 conjuntos disjuntos. En cada subdivisión contamos la cantidad de ejes entre conjuntos y tomamos la subdivisión con menor de ejes. Resolverlo por fuerza bruta, tendria una complejidad exponencial.

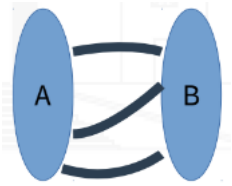


Figure 11: *Ejemplo de k conectividad*

Para solver el problema, vamos a realizar una reducción al problema de flujos:

1. Por cada eje $e = (u, v)$ creamos 2 ejes dirigidos (u,v) y (v,u), les asignamos una capacidad de 1.
2. Por cada combinación posible de dos nodos, etiquetamos como s y t respectivamente y resolvermos *"MAX-FLOW MIN-CUT"*.

3. El menor de los cortes mínimos corresponde al valor de la K-conectividad del eje del grafo.

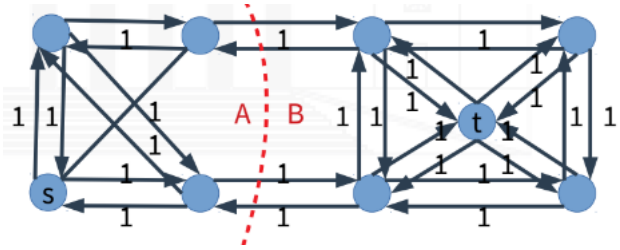


Figure 12: *Ejemplo de k conectividad*

La complejidad temporal va a ser $O(n^5)$ ya que por una lado, vamos a repetir el problema de flujo maximo en $O(|V|^2)$ y por el otro en el peor de los casos, si usamos ford-fulkenson la complejidad sera $O(|V|^3)$.

Algoritmo de Karger: Es un algoritmo randomizado que funciona en tiempo polinomial y *puede retornar un resultado erroneo*. Utiliza un proceso de contracción (*Contraction Algoritms*). El proceso de contracción es el siguiente:

1. Seleccionar: Un eje $e = (u, v)$ de forma aleatoria y uniforme.
2. Reemplazar: Los nodos u y v por un nuevo nodo w. Todos los ejes (u, v) se eliminan. Los ejes (u, a) y (v, a) con $a \in E - \{u, v\}$ se reemplaza por (w, a).
3. Repetir: Hasta que solo queden 2 nodos en el grafo G resultante.

Para encontrar K con alta probalidad de exito hay que ejecutar el algoritmo varias veces y quedarnos con la mejor K entre todas las iteraciones. Esto da una complejidad total temporal de $O(|V|^4 * log|V|)$, funcionando peor y puede fallar con cierta probabilidad. La mejora de **steiner-karger**(1996) aplica la siguiente variante:

1. Si $|V|$ es pequeño resuelve por fuerza bruta
2. Sino contrae $|V|/\sqrt{2}$ nodos y aplica técnica de división y conquista con el resto.

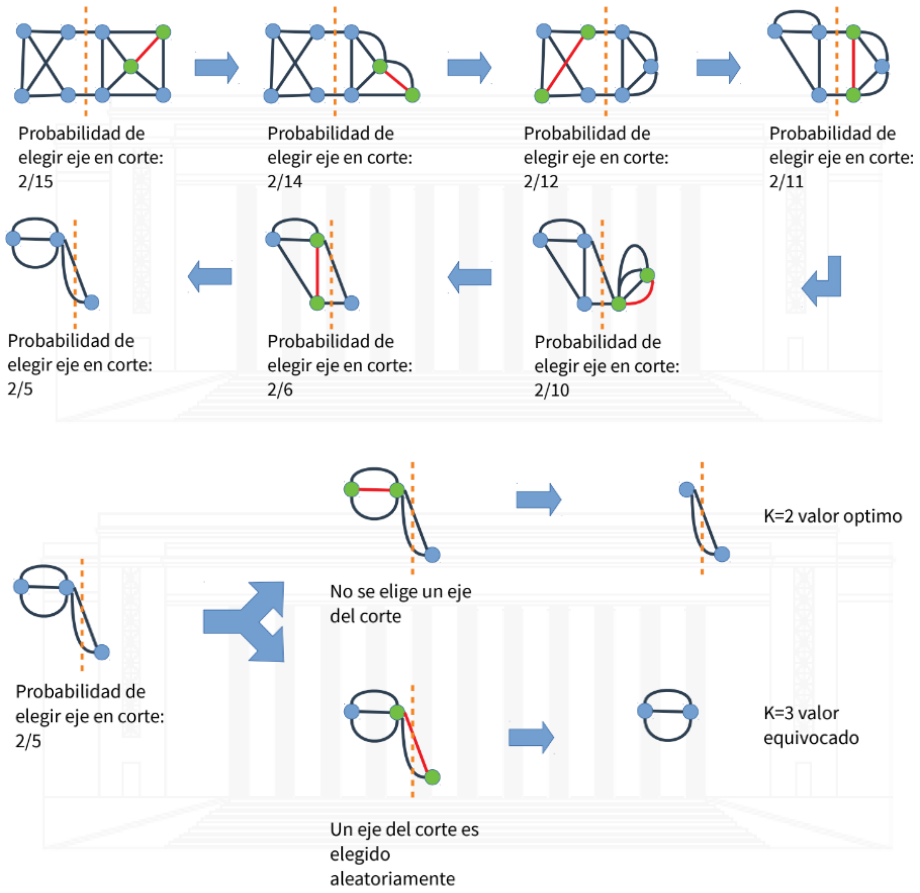


Figure 13: *Ejemplo de contracción Karger*

Con esto encuentra con alta probabilidad K, con complejidad temporal $O(|V|^2 log^3|V|)$.

8.3 Resolución de conflictos en sistemas distribuidos

Supongamos que tenemos n procesos $P_1, P2, ..., P_n$, cada uno compite para acceder a un recurso compartido (ejemplo: base de datos). Divimos el tiempo en rondas discretas. Las bases de datos tienen la propiedad de que esta puede ser accedida por un solo proceso a la vez en una ronda. Si dos o mas procesos intenta acceder a esto simultaneamente, entonces todos los procesos seran lockeados en lo que dura la ronda.

Diseño de un algoritmo randomizado: La aleatoriedad provee un protocolo natural para este problema, el cual se puede especificar de la siguiente manera. Para un valor de $p > 0$, que determinaremos luego, cada proceso intentara acceder a la base de datos en cada ronda con probabilidad p, independiente de la decisión que tomen los otros procesos. Entonces:

1. Si exactamente un proceso solicita el recurso en una ronda, lo conseguira con exito.
2. Si dos o mas procesos solicitan el recuros, se lockearan.
3. Si nadie solicita el recurso, el turno se pierde.

Esta estrategia, se conoce como quiebre de simetria. Si cada vez que un proceso falla, solicita el proceso inmediatamente en el turno siguiente se provocará un atasco, pero utilizando la aleatoriedad se puede corregir la contensión.

ANALISIS PENDIENTE!!

En conclución, con probabilidad de $1 - n^{-1}$ todos los procesos tendran exito en acceder al recurso al menos 1 vez en no mas de $t = 2 * \lceil en \rceil * lnO(n)$ rondas.

8.4 Quick Sort Randomizado

Sea un set $S = \{a_1, a_2, ..., a_n\}$, la mediana es el número que queda en la posición del medio si se presentan ordenados. Formalmente, el mediano de S es igual K-esimo elemento mas grande en S, donde:

- Si n es impar, $K = n/2$.
- Si n es par, $K = (n + 1)/2$.

La complejidad de Resolución hasta aqui es de $O(nlogn)$ debido al ordenamiento.

```
select(S,k)
    Sl={}  Sr={}
    p = calcular_pivot(k)
    Desde j=1 hasta k
        Si sj < p
            Sl += {sj}
        Si sj > p
            Sr += {sj}

    Si size(sl) = k-1
        return p
    Sino si size(sl) > k - 1
        select (sl, k)
    sino
        select (sr, k-1 - size(sl))
```

Listing 23: Algoritmo para calcular la mediana utilizando división y conquista

Analizando la funcion calcular pivot, calculamos un pivot "centrado" que al menos $\epsilon * n$ elementos menores y mayores que él($\epsilon > 0$). Proponemos seleccionar un $a_i \in S$ como **pivot uniforme al azar**, considerando centrales a los elementos que al menos dejen 1/4 de los elementos del lado izquierdo o derecho. La mitad de los elementos son centrales ($\epsilon = 1/4$). La probabilidad de seleccionar un pivot central es de 1/2.

Al dividir en S+ y S- verificamos que la división cumpla el requisito, y sino cumple, volvemos a seleccionar al azar otro pivot. Probabilisticamente tendria que repetir a lo sumo 2 veces la elección. **Este proceso es O(n)**.

QuickSort es un algoritmo de ordenamiento que utiliza división y conquista. Divide en cada paso en dos subproblemas utilizando un valor pivot. Por un lado se procesan los valores menores al pivot y por el otro los mayores.

```
QuickSort(S)
    Si |S| <= 3
        Ordenar S
        Retornar S
    Sino
        p = seleccionar_pivot(S)
        Por cada elemento de S
            Ponerlo en S- si es menor a p
            Ponerlo en S+ si es mayor a p

        S- = QuickSort(S-)
        S+ = QuickSort(S+)

    Retornar S-, p, S+
```

Listing 24: Algoritmo QuickSort

Si el pivot es el valor medio, la complejidad es O(nlog n). Pero si el pivot es malo será $O(n^2)$.

QuickSort Ramdomizado: Modificamos el quicksort intentando **elegir un pivot aleatoriamente**. Al dividir en S- y S+ verificamos que se cumpla el requisito:

1. Queremos que el pivot sea central, ni en 1/4 inicial ni en el final.
2. Si no cumple, volvemos a seleccionar al azar otro pivot.

Probabilisticamente tendria que repetir a los sumo 2 veces la elección del pivot.

```
QuickSort(S)
    Si |S| <= 3
        Ordenar S
        Retornar S
    Sino
        Repetir
        p = seleccionar_pivot(S)
        Por cada elemento de S
            Ponerlo en S- si es menor a p
            Ponerlo en S+ si es mayor a p
        Hasta que |S-| >= 1/4*|S| y |S+| >= 1/4*|S|
        S- = QuickSort(S-)
        S+ = QuickSort(S+)

    Retornar S-, p, S+
```

Listing 25: Algoritmo QuickSort ramdomizado

El proceso total es O(n*log(n)).

9 Algoritmos de aproximación

9.1 Problema del balanceo de carga

Dado un conjunto de maquinas $M_1, M_2, .., M_m$, un conjunto de n tareas donde cada tarea j requiere T_j de tiempo de procesamiento. El objetivo es asignar las tareas a las maquinas de tal forma que la carga quede balanceada (El tiempo asignado a cada maquina sea lo mas parejo posible).

¿Como medimos el balanceo de carga? Si llamamos $A(i)$ al conjunto de tareas asignadas a la maquina i, podemos calcular la carga de la maquina i como:

$$T_i = \sum_{j \in A(i)} t_j$$

Lo que buscamos es minimizar el makespan; esto es simplemente la maxima carga sobre cualquier maquina, $T = max_i t_i$. Podemos medir el balanceo por diferente indicadores:

Makespan: max(Ti) para todas las maquinas

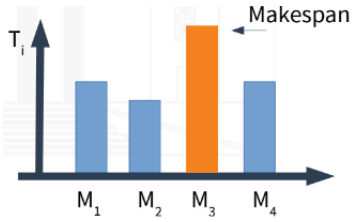


Figure 14: *Ejemplo de makespan*

Un método para seleccionar la asignación y la naturaleza de los trabajos determinará la programación final de las tareas. El problema para la asignación de tareas con minimo makespan es NP-HARD.

Vamos a utilizar un **método greedy** para realizar la aproximación. Para cada tarea i, asignar a la maquina j con menor carga en el momento.

```
Comenzar sin trabajos asignados
Definir Ti=0 y A(i) != 0 para todas las maquinas Mi
Desde j=1 a n
   Sea Mi la maquina con menor Tk (k=1 a m)
```

```
    Asignar la tarea j a maquina Mi
    Establecer A(i) <- A(i) union {j}
    Establecer Ti <- Ti + Tj
```

Listing 26: Algoritmo de aproximación greedy

Analisis: Podemos acotar inferiormente el tiempo optimo para el makespan de dos maneras:

- $T* \geq \frac{1}{m} \sum_j t_j$: El optimo es mayor o igual al tiempo promedio total.
- $T* \geq max_j t_j$: El optimo es mayor o igual al tiempo del trabajo mas largo.

El algoritmo asigna los trabajos a las maquinas con un makespan T $\leq$ 2T*. Demostración pendiente.

En el peor de los casos, si la ultima tarea coincide con aquella de longitud mas grande, quedara peor balanceado.

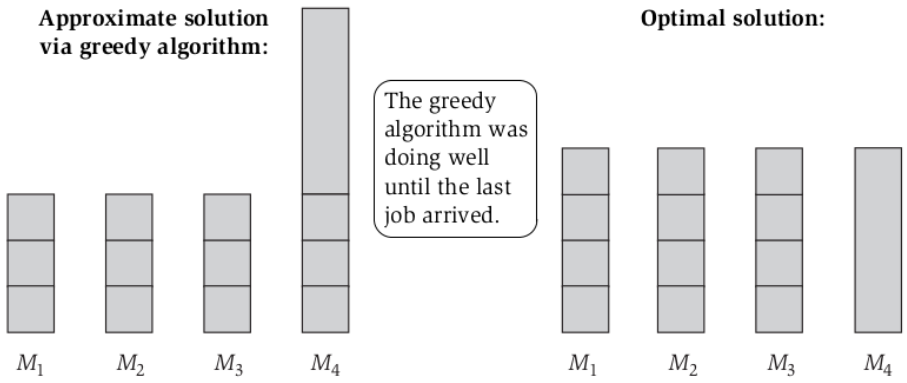


Figure 15: *Ejemplo de balanceo con greddy*

Algoritmo mejorado: Se agrega un preprocesamiento para ordenar las tareas segun su duración en forma decreciente.

```
Comenzar sin trabajos asignados
Ordenar las tarea de mayor a menor duracion
Definir Ti=0 y A(i) != 0 para todas las maquinas Mi
Desde j=1 a n
    Sea Mi la maquina con menor Tk (k=1 a m)
```

```
    Asignar la tarea j a maquina Mi
    Establecer A(i) <- A(i) union {j}
    Establecer Ti <- Ti + Tj
```

Listing 27: Algoritmo de aproximación greedy mejorado

El algoritmo asigna los trabajos a las maquinas con un makespan $\mathbf{T \leq 3/2T^*}$. Demostración pendiente.

9.2 Problema del selección de centros

Tenemos un conjunto de S de n sitios. Queremos seleccionar k centros para construir diferentes shopping malls. Esperamos que cada persona de las n ciudades pueda ir a comprar a uno de los shopping, y queremos seleccionar los sitios de los k shopping centrales.

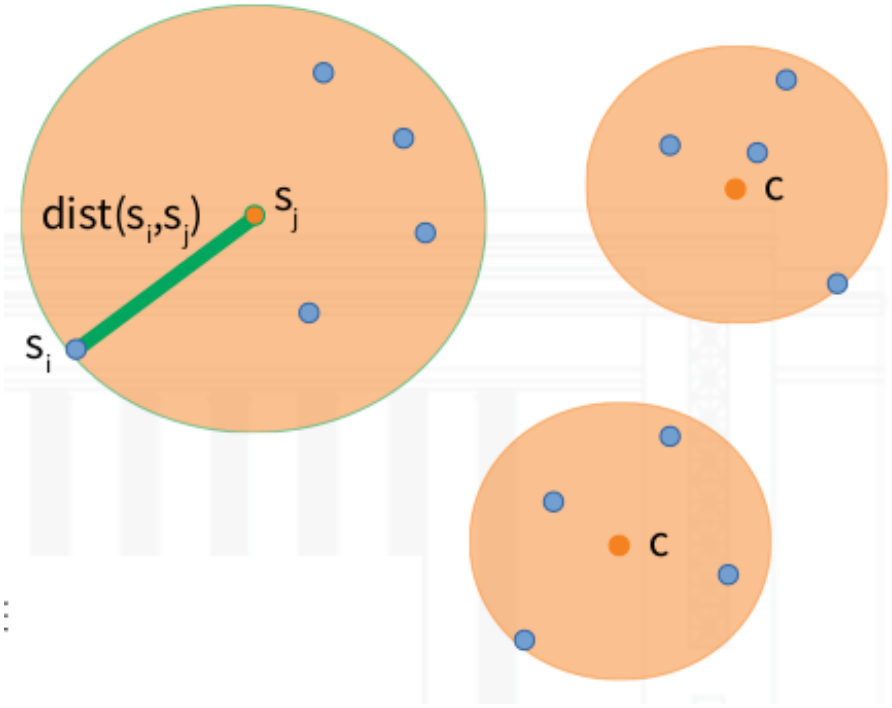


Figure 16: *Ejemplo con k=3 centros*

```
1
2     Asumimos k < |S| (sino definimos C = S)
3     Seleccionar cualquier sitio 's' y convertirilo en un centro C =
       {s}
4     Mientras |C| < k
5         Seleccionar sitio 's' pertenece a S que maximice la
      distancia dist(s,C)
6         C = C U {s}
7     Retornar C como los sitios seleccionados
```

Listing 28: Algoritmo de aproximación greedy

Algoritmo de aproximación greedy que retorna un conjunto C de k puntos tales que r(C) $\leq$ 2r(C*) donde C* es un conjunto optimo de k puntos.

Corresponde a una **2-aproximación** del problema de centros.

9.3 Problema del cobertura de conjuntos

Es un problema ya planeado donde vimos que es NP-Completo, pero al tener grandes aplicaciones podemos aproximarlo para resolverlo en tiempo polinomial. Sea un conjunto X de n elementos y una lista de F=$S_1, ..., S_m$ de subconjuntos de elementos de X.

Un *SetCover* es un subconjunto de F cuya unión es X. El objetivo es encontrar el set cover S con menor cantidad de subset.

Tratando de construir un buen algoritmo construiremos un algoritmo greedy donde iremos seleccionando un subset de F paso a paso para el cover set. Intentaremos cubrir la mayor cantidad de elementos aún no seleccionados.

```
1
2   Empezamos R = X y S = Vacio (Sin conjuntos seleccionados)
3   Mientras R != 0
4       Seleccionar el conjunto Si con mayor puntos cubiertos en R
5       Agregamos Si a S
6       Quitamos elementos de Si de R
```

Listing 29: Algoritmo de aproximación greedy

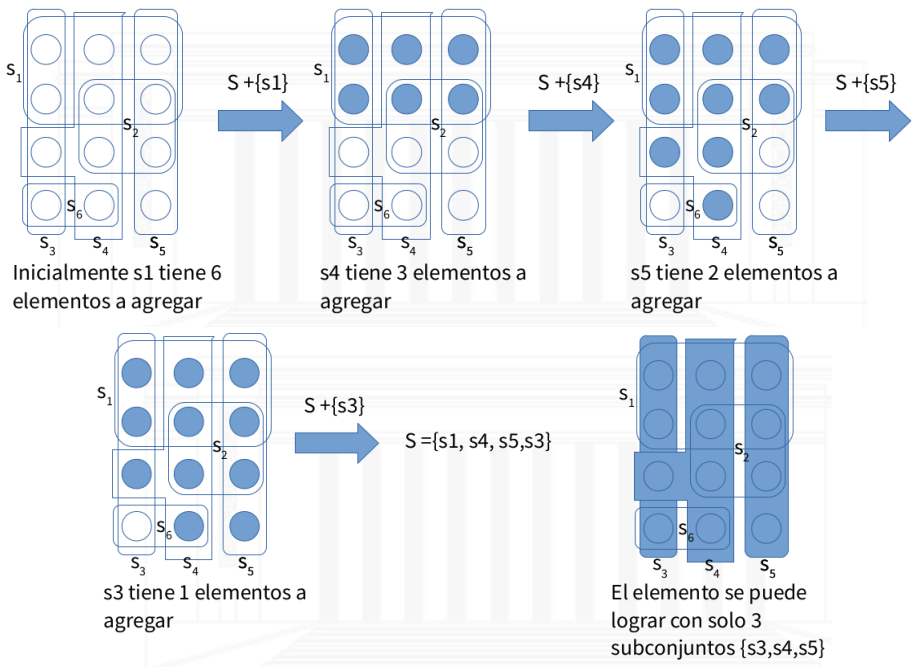


Figure 17: *Ejemplo ejecución de coverset utilizando Greedy. Se puede observar que el algoritmos nos dio una solución que no es la óptima.*

El algoritmo es un **(1+log n)-algoritmo de aproximación**.

9.4 Problema del cobertura de vertices

Sea un grafo $G = (V, E)$ no dirigido, donde cada vértice i tiene un peso $w_i \geq 0$. Queremos encontrar un subconjunto $S \subset V$ donde cada arista de E del grafo pertenezca a algún vértice de S. Minimizando el costo de los vértices seleccionados. Esta caso es mas general que el problema que ya se vio con pesos en los vértices igual a uno.

Costo pagado: Existen diferentes sub conjuntos S de V que conforman un vertex cover. Llamaramos $w(S)$ como el costo del vertex cover formado por $S \subset V$.

La solución optima S* es aquella para la que $w(S*) \leq w(S)$ para todo S.

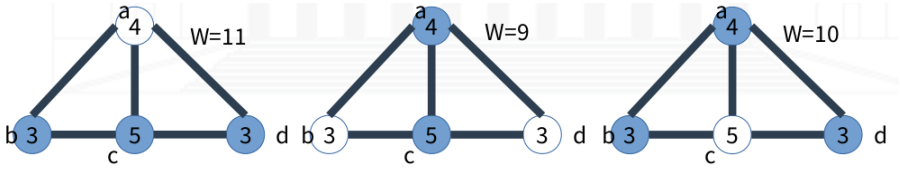


Figure 18: *Se pueden elegir varios subconjuntos de vértices, pero se busca el de menor costo.*

Mediante el método de Pricing, podemos pensar cada peso de los *vértices* como un "costo". Cada *eje* es un "agente" dispuesto a pagar algo al vértice que lo cubre. Entonces se puede ver que los agentes deberian pagar un costo para ser cubiertos por lo menos por uno nodo.

Diremos que un vertice esta *pagado* si la suma de lo pagado por sus ejes es igual al costo de sus vértices.

Diremos que un precio p_e que tiene que pagar un eje e al vertice i es justo si no paga mas que el costo w_i del vertice.

```
1
2   Definir pe = 0 para todo e pertenece a E
3
4   Mientras exista un eje e=(i,j) tal que i o j no este "Pagado"
5       Seleccionar el eje e
6       Incrementar pe si violar la integridad
7
8   Sea S el conjunto de todos los nodos pagados
```

```
9     Retonar S
```

Listing 30: Algoritmo de aproximación greedy. La integridad se respeta si se paga el precio justo.

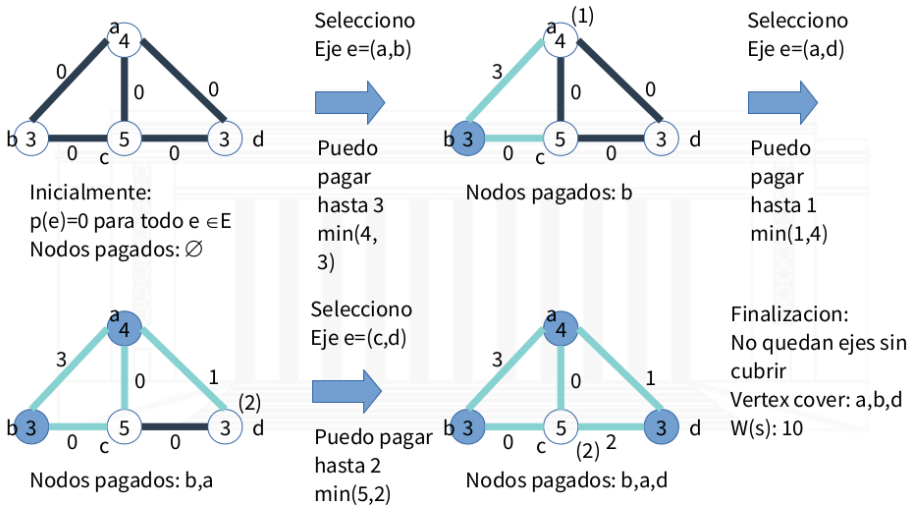


Figure 19: *Inicialmente cada eje no paga nada y no esta cubierto, no hay nodo pagado. selecciono algun eje, por ejemplo el e=(a,b). El eje paga el minimo que es 3. Se marca el nodo b como "pagado" y se cubre el eje (a,b) y (b,c). Luego selecciono e=(a,d) y se cubre los ejes (a,d) y (a,c) y se marca como pagado el nodo a. Por último, selecciono el eje e=(c,d), pagando el minimo que es 2, se cubre el eje (c,d) y se marca como pagado el nodo c. El algoritmo termina diciendo que los vertices cubiertos son a,b,d pagando W(s)=3+4+3=10*

El costo del conjunto S retornado por el algoritmo es como mucho el doble de algún vertex cover posible. El algoritmo es un **2-algoritmo** de aproximación

$$w(S) \leq 2w(S*)$$

9.5 Problema de la mochila

Supongamos que tenemos n elementos que queremos guardar en una mochila. Cada elemento $i = 1, .., n$ tiene dos parametros: un peso w_i y un valor v_i. Siendo W la capacidad de la mochila, el objetivo global del problema es encontrar un subconjunto de S con los elementos que maximicen el valor sujeto a la restricción del peso total que no debe superar W.

Nuestro problema tomara como entrada los pesos, los valores definidos por el problema y tambien tomaremos un parametro extra ϵ, para obtener la precision deseada.

Llamamos $OPT(i, V)$ al subproblema de determinar el menor peso que se puede obtener con los primeros i items cuyo valor iguale o supere el valor de al menos V en la mochila.

```
1
2    Desde i=0 a n
3        OPT[i][0]=0
4
5    Desde v=1 a Vmax
6        OPT[0][v]=+infinito
7
8    Desde i=1 a n   // Elementos
9        Desde v=1 a Vmax // valores
10
11           enOptimo = w[i] + OPT[i-1, v-v[i]]
12           noEnOptimo = OPT[i-1, v]
13
14           si enOptimo < noEnOptimo
15               OPT[i][v] = enOptimo
16           sino
17               OPT[i][v] = noEnOptmimo
18
19   Desde v=Vmax a 0
20       si OPT[n,v] <= w
21           retornar OPT[n][v]
```

Listing 31: Algoritmo de aproximación con programación dinamica usando interativo

La complejidad temporal es $O(nV_{max})$ y la espacial es $O(nV_{max})$. Para recobrar los elementos seleccionados debemos almacenar para cada caso si se selecciono o no que él elemento este en el óptimo. Iterar desde el óptimo hacia

atrás reconstruyendo.

Si llamamos v*=max v_i con $0 < i \leq n$ podemos acotar:

$$V_max = \sum_{j=1}^{n} v_j \leq nv*$$

Por lo tanto la complejidad de la programación dinámica sera $O(n^2, v*)$. Si n es pequeño el algoritmos se resuelve en tiempo polinomia. Sino, se resolvera en tiempo pseudo polinomial.

Para realizar la aproximación, utilizaremos un parametro b de rendondeo, donde para cada elemento i calculamos:

$$\tilde{v_i} = \lceil v_i/b \rceil b$$

De esta forma queda acotado $v_i \leq \tilde{v_i} \leq v_i + b$ y ademas es multiplo de b. Podemos resolver mediante programación dinamica $v'_i = \lceil v_i/b \rceil$. De esta forma, nos quedara un Vmax mas pequeño, una cota v* mas pequeño, **la programación dinamica sera mas manejable y se ejecutara como si fuese polinomial.**

Para elegir b utilizaremos:

- ϵ para generar b
- con $0 < \epsilon \leq 1$
- Y por comodidad $\frac{1}{\epsilon} = \epsilon^{-1}$ es un *número entero.*

Un valor conveniente para $b = \epsilon v * /2n$.

```
1 Knapsack-Approx(e:epsilon)
2     Obtenemos el precio maximo vmax
3     b = (e/2n) * vmax / 2n
4     Para cada elemento i
5         Calcular v'i con b
6
7     Resolvemos con programacion dinamica con los valores v'i
8
```

```
9     Retornar el conjunto de elementos encontrados.
```

Listing 32: Knapsack-Approx(e:epsilon)

En todo el proceso la **complejidad temporal global** sera $O(n^3\epsilon^{-1})$ para un valor fijo de e el algoritmo se ejecuta en tiempo polinomial.

Para cualquier $\epsilon > 0$, la solución aproximada encuentra una solución factible cuyo valor esta dentro de un factor $1 + \epsilon$ de la solución óptima.

10 Teoría de la computabilidad

La teoria de la computabilidad comienza con una pregunta. ¿Que es una computadora?. Se utiliza una computadora ideal llamado modelo computacional.

10.1 Automata finito

	0	1
q_0	q_1	q_1
q_1	q_1	q_2
q_2	q_1	q_3
q_3	q_1	q_1

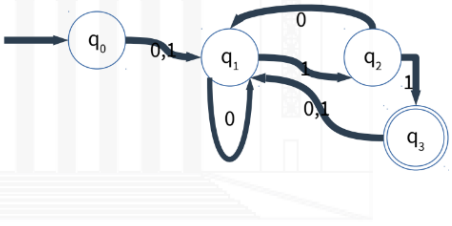


Figure 20: *Ejemplo de tabla de función de transición y diagrama de estados*

Definición formal de automata finito: Un autómata finito "M" es una 5-tupla $(Q, \sum, \delta, q_0, F)$ donde:

1. Q Conjunto finito llamado "estados". (Ejemplo: q_0, q_1, q_3)
2. $\sum$ Conjunto finito llamado "alfabeto" (ejemplo: 0, 1)
3. $\delta : Q \times \sum \to Q$ es una función de transición.
4. $q_0 \in Q$ estado inicial (ejemplo q_0).
5. $F \subset Q$ conjunto de estados de aceptación. (ejemplo q_3)

El autómata recibe un string de entrada (escrito en el alfabeto $\sum$), procesa el string partiendo del estado inicial utilizando la función de transición y produce una salida:

- "Aceptación": si al terminar de procesar el string el estado final corresponde a uno de aceptación.

- "Rechazo" si no es de aceptación.

Definición formal de computo: Sea $M = (Q, \sum, \delta, q_0, F)$ un autómata finito y dado $w = w_1, w_2, \ldots, w_n$ una cadena donde cada w_i es parte del alfabeto $\sum$, entonces M acepta w si existe una secuencia de estados $r_0, r_0, \ldots, r_n$ en Q con las condiciones:

1. $r_0 = q_0$
2. $\delta(r_1, w_{i+1}) = r_{i+1}$, para cada $i = 0, \ldots, n-1$ y
3. $r_n \in F$

La *condición 1* dice que la maquina de estado comienza en el estado de comienzo. La *condición 2* dice que la maquina de estado para de un estado a otro deacuerdo a la función de transición. La *condición 3* dice que la maquina acepta su entrada si este termina en un estado aceptado. Decimos que M reconoce el lenguaje A si $A = \{w | M \text{ acepta a } w\}$.

10.2 Automata finito no determinista (AFND)

En una maquina no deterministica puede haber varias opciones para el siguiente paso en cualquier momento. Al computar una cadena, se van generando en paralelo diferentes ramificaciones de ejecución.

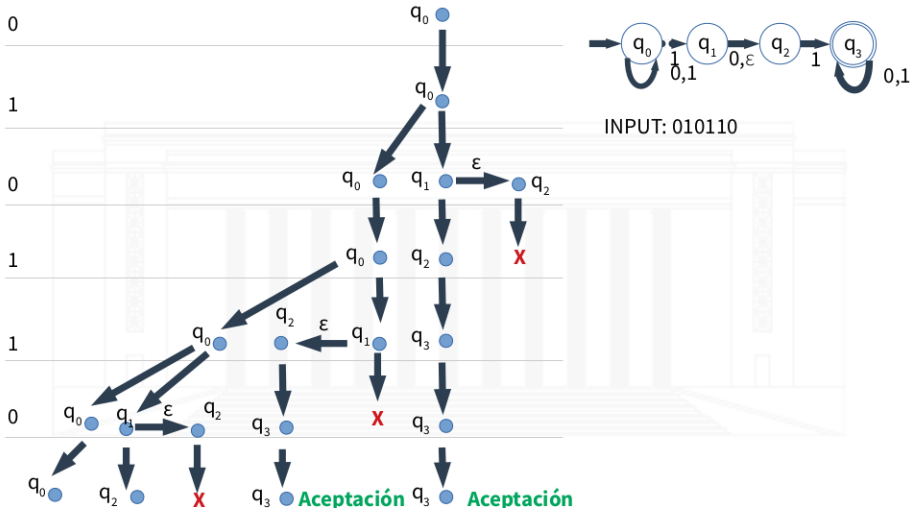


Figure 21: *Ejemplo de automata finito no deterministico*

Al finalizar el procesamiento de una cadena, si al menos una rama termina en estado "Aceptación", retorna "Aceptación". De lo contrario retorna "Rechazo".

A medida que se procesan las ramas pueden abandonar ramas que llegan a estados siguiente $\emptyset$.

Diremos que dos maquinas son equivalentes si reconcen el mismo lenguaje.

10.3 Lenguajes regulares

Un lenguaje es regular si existe algún autómata finito que lo reconozca.

Existen algunas *operaciones* que se pueden aplicar a los lenguajes regulares llamados operaciones regulares llamados *operaciones regulares* cuyo resultado es tambien un lenguaje regular. Podemos utilizar los operadores regulares para construir expresiones que describan lenguajes, los cuales llamaremos ***expresiones regulares***. Ejemplo:

$$(0 \cup 1)0*$$

El valor de esta expresión regular es un lenguaje que consiste en todas las posibles cadenas que empiezan con 0 ó 1, seguido de cualquier cantidad de 0s.

Operadores regulares:

Sean A, B dos lenguajes regulares:

- Union: $A \cup B$
- Concatenación: $A \cup B$
- Estrella $A*$

Definición de expresión regular: Se dice que R es una expresión regular si R es:

1. a para algun a en el alfabeto de $\sum$.
2. ϵ
3. $\emptyset$
4. $R_1 \cup R_2$, donde R1 y R2 son expresiones regulares.
5. $R_1 \circ R_2$, donde R1 y R2 son expresiones regulares.
6. R_1*, donde R1 es una expresión regular.

Ejemplo para convertir una expresión regular $(a \cup b) * aba$ a una AFND. Pasos:

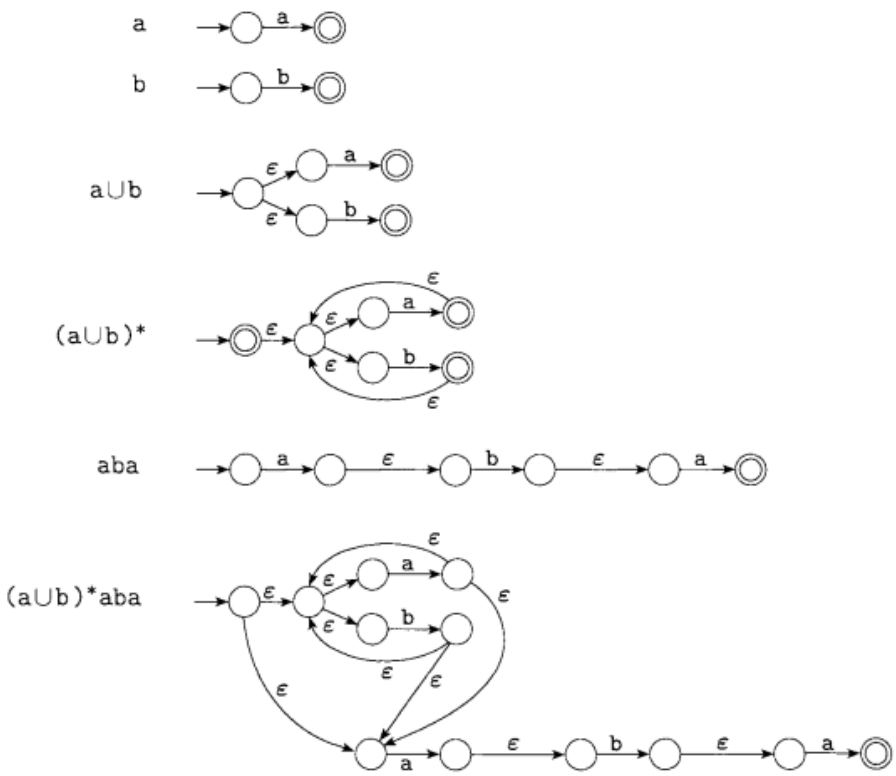


Figure 22: *Ejemplo de expresion regular a AFND*

10.4 Maquina de Turing

Es similar a un automata finito pero con memoria ilimitada y sin restricciones. El modelo de una maquina de Turing, utiliza un *cinta infinita* con memoria ilimitada. Tiene un cabezal de cinta que puede leer y escribir simbolos sobre la cinta. Inicialmente la cinta contiene una cadena de entrada y el resto esta en blanco. La maquina continua procesando hasta que decide producir una salida. Las salida *aceptado* y *rechazado* se obtienen ingresando los correspondientes estados de aceptación y rechazo.

Diferencias entre un automata finito y una maquina de Turing:

1. Puede grabar o leer en la cinta en cada secuencia.

2. La lectura y escritura puede moverse de izquierda a derecha.
3. La cinta es infinita.
4. Los estados especiales para rechazar y aceptar toman efecto inmediatamente.

Configuación

Sucesión de configuraciones

Computo

Lenguaje Turing Reconocible

Loop en una maquina de Turing

10.5 Variantes de Máquinas de turing

1. Con posibilidad de no avanzar ademas de retroceder o avanzar el cabezal.
2. Con multiples cintas.
3. No deterministica.
4. Con impresora donde solo grabo simbolos.

Todas las variantes tienen el mismo poder de computo. Pueden reconocer los mismos lenguajes.

Equivalencia entra una Turing Machine (TM) y una Turing Machine Multicinta(TMM).

Turing Machine No Deterministica (NDTM): Permite en un punto del computo transicionar en simultaneo a varias posibilidades. En un arbol de ramificaciones, alguna rama se puede quedar loopeando en infinito. El recorrido del arbol debe realizarse mediantes BFS para evitar el loopeo.

FALTA EJEMPLO.

En el ejemplo, la maquina de turing multicinta y la no deterministica tienen el mismo poder de computo. O sea, que puede reconocer exactamento los mismos lenguajes, y pueden resolver los mismos problemas.

10.6 Tesis Church-Turing

Entscheidungs Problem: Es un problema de decisión para encontrar un algoritmo general que decidiese si una fórmula del calculo de primer orden es un teorema.

Turing propone su maquina automatica como herramienta de cálculo. Alonzo Church desarrolla su cálculo lambda para el mismo motivo.

Con la Tesis Church-Turing, se dio el marco para definir que es un algoritmo. Pero al ser una tesis no esta probado.

El **10mo** problema planteado por David Hilbert, si lo restringimos a 1 variable, es TURING DECIDIBLE. Pero volviendo al problema original de decidir si una ecuación polinomica diofantica (2 o mas variables) tiene raiz entera, es TURING RECONOCIBLE, pero no es TURING DECIDIBLE.

10.7 Lenguajes Turing no decidibles

Maquina de Turing universal: Permite simular cualquier otra maquina de Turing con un input arbitrario. El input de esta maquina universal, es la maquina a simular y su input.

Existen lenguajes no decidibles:

- Halting Problem
- Post Correspondence problem
- Wang tiles
- Conway's Game of Life
- 10mo problema de Hilbert
- Otros

10.8 Lenguajes Turing no reconocibles

Pueden existir *infinitos leguajes a reconocer* e *infinitas maquinas de Turing* que se pueden generar.

¿Pueden existir lenguajes no reconocibles? Primero vamos a analizar el *tamaño de los infinitos*(Georg Cantos).

Correspondencia entre conjuntos infinitos: Dos conjuntos infinitos tienen el mismo tamaño si pueden establecer entre ellos una *correspondencia biyectiva*.

Conjuntos contables: Un conjunto contable A es contable si es finito o si su tamaño es igual al conjunto de los números naturales. Ejemplo: pares, impares, primos, compuestos. Los números enteros con contables si podemos realizar una correspondencia utilizando algun pivoteo. En el caso de números racionales, utilizando un recorrido tipo espiral, podemos listar los números y realizar la correspondencia con los números naturales. Por lo tanto, los numeros racionales son contables. Pero el conjunto de los números reales no son contables, porque no se puede armar una correspondencia. Al igual que los números binarios no son contables.

El conjunto de las Maquinas de Turing es contable ya que puede realizarse una correspondencia con los números naturales. El conjunto de todos los lenguajes no es contable. Y como un lenguaje es reconocible si una maquina puede reconocerlo. Entonces:

Existen lenguajes no reconocibles por una maquina de Turing

10.9 Complejidad algorítmica con máquinas de Turing

Clase complejidad temporal: Sea $t : N \to R+$ una función, definimos una clase de complejidad temporal TIME(T(n)) a la colección de todos los lenguajes que son decidibles por una maquina de $O(t(n))$-tiempo Maquina de Turing.

Clase complejidad temporal P: P corresponde a la clase de lenguajes que son decibles en tiempo polinómico utilizando una máquina de Turing *deterministica* con cinta única.

$$P = \cup_k TIME(n^k)$$

Clase complejidad temporal NP: P corresponde a la clase de lenguajes que son decibles en tiempo polinómico utilizando una máquina de Turing *no deterministica* con cinta única.

Definimos NTIME(t(n))=L—L es un lenguaje decido por un O(T(n))-tiempo MT no determínisca

$$P = \cup_k NTIME(n^k)$$

NP: nodeterministic polynomial time

Clase complejidad temporal NP-Completo: Un lenguaje B es NP-completo si satisface 2 condiciones:

1. B pertenece a NP, y
2. Todo A que pertenece a NP se puede reducir en tiempo polinomial a B.

10.10 Teorema de Levin Cook

Este teorema permite identificar al problema SAT como el primer lenguaje NP-Completo.