Apunte Final

Técnicas algorítmicas

Backtracking

Optimización combinatoria: Es un problema de optimización cuya región factible es un conjunto definido por consideraciones combinatorias. Por ejemplo, regiones factibles dadas por todos los subjconjuntos / permutaciones de un conjunto finito de elementos, *todos los caminos de un grafo*.

Un algoritmo de **Fuerza Bruta** para un problema de *optimización combinatoria* consiste en generar todas las soluciones factibles y quedarse con la mejor. Suele ser fácil de implementar, y es exacto, es decir, si hay solución, siempre la encuentra.

El Problema de estos algoritmos es la complejidad la cual suele ser exponencial.

Problema de la Mochila

Determinar qué objetos debemos incluir en la mochila sin excedernos del peso máximo **C**, de modo tal de **maximizar** el beneficio total entre los objetos seleccionados.

```
Mochila(S,k):
    if k = n+1
        if peso(S) ≤ C and beneficio(S) > beneficio(B):
            B = S
    else if peso(S) ≤ C and beneficio(S) + sum_beneficios_desde(k+1) > benefi
        Mochila(S u {k}, k+1)
        Mochila(S, k+1)
```

La recursión comienza con B = ø y Mochila(ø,1)

Resolución de Sudoku

```
Sudoku(Tablero, i, j):
    if i = n+1 and j = n+1:
        Chequeo_filas_columnas(Tablero)
    else:
        Para k = 1 hasta 9
            Si fila(i) contiene k: siguiente
            Si columna(j) contiene k: siguiente
            Si cuadrado(i,j) contiene k: siguiente
            Tablero[i][j] = k
            (i',j') = avanzar(i,j)
            Sudoku(Tablero, i', j')
```

Programación Dinámica

La idea es dividir el problema en subproblemas de tamaños menores que se resuelvan recursivamente.

Superposición de Estados: El árbol de llamadas recursivas resuelve el mismo problema varias veces. Con la programación dinámica evitamos este problema con 2 enfoques:

- **Top-Down:** Se implementa recursivamente, pero se guarda el resultado de cada llamada recursiva en una estructura de datos (memorización). Si una llama recursiva se repite, se toma el resultado de esta estructura.
- **Bottom-Up:** Resolvemos primero los subproblemas más pequeños y guardamos todos los resultados.

Problema combinatorio $\binom{n}{k}$

```
combinatorio(n,k):
    Para i = 1 hasta n: A[i][0] = 1
    Para i = 1 hasta n: A[i][i] = 1
    Para i = 2 hasta n:
        Para j = 1 hasta min(i-1, k):
            A[i][j] = A[i-1][j-1] + A[i-1][j]
    retornar A[n][k]
```

Problema del cambio

Supongamos que queremos dar el vuelto a un cliente usando el mínimo número de monedas posibles, utilizando de 1, 5, 10, 25 centavos. Por ejemplo, si el monto es 0,69, deberemos entregar 8 monedas: 2 monedas de 25 centavos, una de 10, una de 5 centavos y cuatro de 1 centavo.

$$f(s) = \begin{cases} 0 & si \quad s = 0 \\ min_{i:a_i \leq s}(1 + f(s - a_i)) & a_k \leq s \\ \infty & caso\ contrario \end{cases}$$

Siendo a_k la moneda de mayor valor, se busca la moneda tal que minimice la cantidad de monedas usadas.

Problema de la mochila

Donde tenemos k objetos y una mochila con capacidad D. Con el llamado $m(n, C)$ obtenemos la solución en una complejidad de O(nC)

$$m(k, D) = \begin{cases} 0 & si \quad k = 0 \\ 0 & si \quad D = 0 \\ max(m(k-1, D), b_k + m(k-1, D - p_k)) & caso\ contrario \end{cases}$$

Greedy

Heurística: Es un procedimiento computacional que intenta obtener soluciones de buena calidad para un problema, intentando que su comportamiento sea lo más preciso posible.

Decimos que *A* es un algoritmo ϵ-aproximado $(\epsilon \geq 0)$ para un problema si su solución *x* respeta que $\left| \frac{x_A - x_{OPT}}{x_{opt}} \right| \leq \epsilon$.

Idea: Construir una solución seleccionando en cada paso la mejor alternativa, sin considerar las implicancias de esta selección

Problema de la mochila

Donde:

- b_i es el beneficio del objeto *i* de *n*
- p_i es el peso del objeto *i* de *n*
- p_i es el peso del objeto i de *n*
- C es la capacidad de la mochila

Queremos que se maximice $\frac{b_i}{p_i}$

```
L = C
i = 1
Mientras L > o and i ≤ n:
    x = min{1, L / pi}
    Agregar una fracción de x del objeto a i a la solución
    L = L - x * pi
    i++
```

Problema del cambio

Seleccionar la moneda de mayor valor que no exceda la cantidad restante por devolver, agregar esta moneda a la lista de la solución, y restar la cantidad correspondiente a la cantidad que resta por devolver hasta que sea 0.

a_i es la i-esima moneda de *k* monedas y t el valor del cambio

```
s = 0
i = 1
Mientras s < t and i ≤ k:
    c = (t-s) / ai
    Agregar c monedas de tipo i a la solución
    s = s + c * ai
    i++
```

En este caso el algoritmo no soluciona ya que hay casos donde no elige la solución optima.

Pero, si existen monedas $m_2, \ldots, m_k$ tales que $a_i = m_{i+1}a_{i+1}$ para $i = 1, \ldots, k-1$ entonces toda solución usa t/a_1 moendas de tipo a_1.

Problema de Tiempo de espera total en un sistema

Un servidor tiene *n* clientes para atender, y los puede atender en cualquier orden. Para $i = 1, \ldots, n$, el tiempo necesario para atender al cliente i es $t_i \in \mathbb{R}_+$. El objetivo es determinar en qué orden se deben atender los clientes para minimizar la suma de los tiempos de espera de los clientes.

Si $I = (i_1, \ldots, i_n)$ es una permutación de los clientes que representa el orden de atención, entonces la suma de los tiempos de espera es $T = t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) = \sum_{k=1}^{n}(n-k)t_{i_k}$.

Algoritmo: En cada paso, atender al cliente pendiente que tenga menor tiempo de atención. Retorna $I_{Greedy} = (i_1, \ldots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \ldots, n-1$

Intro grafos

Orden topológico en DAGs

```
topological_sort(graph G) {
    dfs(G) // Guardando el start y finish de cada nodo
    return reversed(finish)
}
```

Todas las aristas van de menor a mayor $\iff \forall u, v / (u, v) \in E(G) : finish[u] > finish[v]$

Demostración:

Podría existir otro camino que conecta y y v, entonces:

1. $start[u] < start[v]$
   Por cualquier camino va a valer que
   $finish[u] > finish[v]$
2. $start[u] > start[v]$

   Esto no puede ocurrir porque los DAGs no tienen ciclos

Kosaraju (c. f. c.)

```
Kosajaru(graph G):
    G_t = invertir(G)
    dfs(G_t)
    dfs(G) // Usando finish de G_t en sentido inverso
```

Árboles Generadores Mínimos

Equivalencias en árboles

1. G es un **árbol**
2. G es un grafo sin circuitos simples y e una arista tal que $e \notin E(G)$. $G + e = (V, E + \{e\})$ tiene exactamente un **único** circuito simple, y ese circuito **contiene** a e.
3. $\exists$ exactamente un camino simple entre todo par de nodos.
4. G es conexo, pero si se elimina cualquier arista se desconecta, es decir toda arista es **puente**

Demostraciones

$1 \rightarrow 2)$ Como G es conexo $\rightarrow$ Sea $e = (u, v) / e \notin E(G)$ entonces existe algún camino simple P_{uv} entre u y v. Por lo tanto se forma un circuito $C : P_{uv} + e$ en $G + e$. Supongamos que existen 2 circuitos, $C^P : P_{uv} + e$ y $C^Q : Q_{vu} + e$ en $G + e$, entocnes existe algún circuto simple $C' : P_{uv} + Q_{vu}$ en $G + e$ que no usa a e por lo tanto también existe en G lo cual es **absurdo**.

$2 \rightarrow 3)$ Sean u y v vértices distintos de G. Si $e = (u, v)$ una arista de G entonces e es un camino simple entre u y v. Si no existe algún circuito simple $C : P_{uv} + e$ en $G + e$. Entonces existe algún camino simple P_{uv} entre u y v en G. Por lo tanto existe P_{uv} entre todo par de vértices u y v. No existen más porque: igual a la de arriba.

$3 \rightarrow 4)$ Existe un camino simple P_{uv} entre todo par de vértices u y v entonces G es conexo, y como este camino es **único** si saco cualquier arista $e \in P_{uv}$ se desconecta el grafo.

$4 \rightarrow 1)$ G es conexo, si existe un circuito simple C en G y $e = (u, v) \in C$, si saco e no se desconecta lo cual es **absurdo** ya que todas las aristas de G son puentes, por lo tanto no existe C.

Árbol Generador

1. Todo G conexo tiene al menos un AG

   Demostración: Mientras haya un ciclo en el grafo, se saca una arista cualquiera de ese ciclo hasta que no queden más ciclos. El grafo resultante es conexo y sin ciclos, entonces es AG de G.

2. Si G conexo tiene **un sólo** AG entonces es un árbol

   Demostración: Sean T y T' AG distintos de G y $e = (u, v) \in T' \setminus T$, $T + e$ tiene un ciclo, por lo tanto G tiene un ciclo, lo cual es **absurdo** porque G es un árbol.

3. $T = (V, E_T)$ es AG de $G = (V, E)$. Sea $e \in E \setminus E_T$ tal que $T' = T + e - f = (V, E \cup \{e\} \setminus \{f\})$ con f una arista distinta de e del único circuito simple que se forma al agregar e a $T \rightarrow T'$ es otro AG de G.

   Demostración: $T = (V, E_T)$ es AG de $G = (V, E)$. Sea $e \in E \setminus E_T$ y $f \neq e \in C$, siendo C el **único** circuito simple que se en $T + e$. $T' = T + e' f = (V, E \cup \{e\} \setminus \{f\})$, T' es conexo ya que tiene *n-1* aristas y no tiene circuitos simples. T' tiene los mismos vértices que G, por lo tanto es subgrafo generador. T' tiene n-1 aristas, entonces es AG de G.

Árbol Generador Mínimo

Elección golosa: En cada paso se busca $e = (u, v)$ tal que $u \in S$ y $v \in V \setminus S$ y $w(e)$ es el mínimo de las aristas que cruzan el corte

Demostración:

Sea T AGM de G y $e = (u, v) \in E(G)$

- Si $e \in T$ listo
- Si $e \notin T$ elijo $f = (x, y), x \in S, y \in V \setminus S$ y $f \in P_{uv}$,

  entonces $T' = T - f + e$ también es AG de G por lo visto antes.

  $w(T') = w(T) - w(f) + w(e)$, como $w(e) \leq w(f)$ **(elección golosa)**

  $w(T') = w(T) - w(f) + w(e) \leq w(T)$ por lo tanto T' es AGM

Prim

```
edges prim(node v, graph G) {
    tree_edges = ø
    tree_nodes.agregar(v)
    i = 1
    Mientras i ≤ n - 1:
        e = min(w(e), e = (v,u), v in tree_nodes and u in nodes - tree_nodes)
        tree_edges.agregar(e)
        tree_nodes.agregar(u)
        i++
```

```
    retornar tree_edges
}
```

Complejidad

- Min-Heap: $O(E \cdot log(V))$
- Fibonacci-heap: $O(V \cdot log(V) + E)$

Correctitud

Dado $G = (V, E)$ un grafo conexo. $T_k = (V_{T_k}, E_{T_k}), 0 \leq k \leq n-1$, es **árbol y subgrafo de un árbol generador mínimo de G**

Demostración:

Caso base: Antes de ingresar al ciclo, *k = 0,* $T_0 = (\{v\}, \emptyset)$ **es** árbol y subgrafo de todo AGM de G.

Paso inductivo: Para demostrar el paso inductivo consideramos $T_k, k \geq 1$

La *hipotesis inductiva* es:

$T_{k'}, k' < k$ **es** árbol y subgrafo de algún $T = (V, E_T)$ AGM de G.

Llamaos w al vértice agregado en la iteración k y e a la arista. Es decir, $T_k = (V_k, E_k)$ donde $V_k = V_{k-1} \cup \{w\}$ y $E_k = E_{k-1} \cup \{e = (u, w)\}, u \in V_{k-1}, w \notin V_{k-1}$.

Por **HI** sabemos que T_{k-1} es árbol. Como T_k tiene un vértice y una arista más que T_{k-1} y es conexo, entonces T_k es árbol. También sabemos por **HI** que existe $T = (V, E_T)$ AGM de G tal que T_{k-1} es subgrafo de T.

- Si $e \in E_T$, es decir si T contiene a e, entonces T_{k+1} también es subgrafo de T.
- Si $e \notin E_T$, entonces $T + e$ tiene un circuito simple $C = P_{uw} + e$, que contiene a e. Este circuito está formado por el único camino entre u y w que tiene T, P_{uw} más la arista e. Sea f una primera arista de P_{uw} que tiene un estremo en V_{k-1} y el otro no en V_{k-1}. Definimos $T' = T + e - f = (V, E_T \cup \{e\} \setminus \{f\})$:
  - T' es árbol generador de G
  - T_k es subgrafo de T'
  - T' es AGM de G:

    Como f es una arista elegible al comienzo de la iteración k, pero el algoritmo eligió a e, seguro cumple que $w(f) \geq w(e)$. Entonces,

    $w(T') = w(T) + w(e) - w(f) \leq w(T)$

Con esto demostramos que T_k es árbol y subgrafo de un AGM de G.

Al finalizar la iteración n-1, T_{n-1} es árbol y subgrafo de algún $TAGM$ de G. Además, T_{n-1} es subgrafo generador de G, ya que en cada iteración el algoritmo agrega un vértice distinto a V_T, entonces $V_{n-1} = V$.

Por lo tanto, T_{n-1} es T, AGM de G

Kruskal

```
edges kruskal(graph G) {
    sort(edges(G))
        tree_edges = ø
        i = 1;
        Mientras i ≤ n - 1:
            e = min(edges)
            edges.eliminar(e)
            Si componente(e.v) ≠ componente(e.u):
                tree_edges.agregar(e)
                i++
        retornar tree_edges;
}
```

Complejidad

- $O(E \cdot log(V))$

Correctitud

Dado $G = (V, E)$ un grafo conexo. $T_k = (V, E_{T_k}), 0 \leq k \leq n - 1$, es B**de un árbol generador mínimo de G**

Demostración:

Muy similar a la demo de **prim**.

Si $e = (w, v)$ es la arista incorporada en k-ésimo lugar, la única diferencia significativa es la definición de la arista f en la demostración del paso inductivo cuando $e \notin T_{k-1}$.

Llamamos G_u^{k-1} a la componente conexa de u antes de agregar la arista e.

La arista f en la definición de $T' = T + e - f$ debe ser una arista que pertenezca al circuito $C = P_{wu} + e$ y que tenga un extremo en G_u^{k-1} y el otro extremo fuera de G_u^{k-1}. Esta arista seguro existe, porque u pertenece a esa componenete y w no pertenece. Esta arista f es candidata a ser elegida por el algoritmo en lugar de e, por lo que $w(f) \geq w(e)$.

Camino mínimo

Dijkstra

```
void dijkstra(nodo v) {
    S = ø
    pred[v] = 0
    dist[v] = 0

    Para u en V:
        e = (v,u)
        Si e in E:
            pred[u] = v
            dist[u] = w(e)
        Si no
            pred[u] = INF
            dist[u] = INF

    mientras S ≠ V:
        w = min(dist[u],u in V\S)
        S.agregar(w)
        Para u in V\S and (w,u) in E:
            Si dist[u] > dist[w] + w((w,u)):
                pred[u] = w
                dist[u] = dist[w] + w((w,u))
}
```

Complejidad

- $O(E \cdot log(V))$ o $O(V^2)$

Correctitud

- Dado un digrafo $G = (V, E)$ con pesos positivos en las aristas, al finalizar la iteración k el algoritmo de *Dijkstra* determina, siguiendo hacia atrás **pred** hasta llegar a *v*, un camino mínimo entre el vértice *v* y cada vértice *u* de S_k, con longitud $d(v, u)$ (Siendo S_k el valor del conjunto de S al finalizar la iteración k).

Demostración: Haciendo inducción en las iteraciones, k.

Caso base: Antes de entrar por primera vez al ciclo, $k = 0$, vale que $S_0 = \{v\}$ y $d(v, v)$ = 0. Esto es correcto ya que el camino mínimo desde v hasta v es 0 por definición.

Paso inductivo: Consideramos una iteración $k', k \geq 1$.

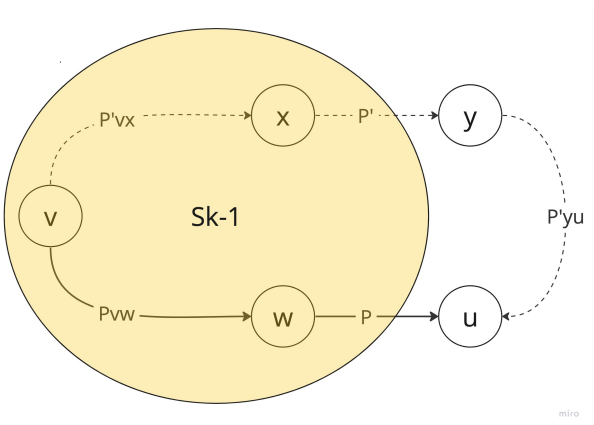
La hipótesis inductiva es: Al terminar la iteración k' $(k' < k), \forall u \in S_{k'}, d(v, u)$ es **la longitud de un camino mínimo de v a u finalizado en $pred[u]$.**

Sea u el vértice agregado a S en la iteración k, $S_k = S_{k-1} \cup \{u\}$, con $pred[u] = w$. Como $w \in S_{k-1}$, por **HI**, desde v a w (w($P_{vw} = d(v, w)$)).

Definimos $P = P_{vw} + w(w \to u)$. Sea P' otro camino de v a u y y el primer vértice de ese camino que no está en S_{k-1} (debe existir porque $v \in S_{k-1}$ y $u \notin S_{k-1}$) y x el inmediatamente anterior.

Si y es u, entonces $w(P') = d(v,x) + w(x \to u)$. Si x ingresó a S después que el u, en ese momento el algoritmo comparó $d(v,u)$ (que era $d(v,w) + w(w \to u)$ por ser $pred[u] = w$) con $d(v,x) + w(x \to u)$ y no acutalizo $pred[u]$, por lo que asegura que $d(v,x) + w(x \to u) \geq d(v,u)$. Si x ingresó a S antes que w, al momento de ingresar a w actualizó $pred[u]$, lo que también asegura que $d(v,x) + w(x \to u) \geq d(v,u)$. Demostrando que $w(P) \leq w(P')$.

Ahora supongamos que y n es u. Tenemos el siguiente esquema:



En la iteración k los vértices y y u son candidatos a entrar a S y el algoritmo elige a u, lo que implica que $d(v,w) + w(w \to u) = d_{k-1}(v,u) \leq d_{k-1}(v,y)$.

Como $x \in S_{k-1}$, hace esta comparación cuando ingresó x $xS : d_{k-1}(v,y) \leq d(v,x) + w(x \to y)$ y por **HI**, $d(v,x)$ es la longitud de un camino mínimo desde v a x, tenemos: $w(P) = d(v,w) + w(w \to u) =$

$d_{k-1}(v,u) \leq d_{k-1}(v,y) \leq d(v,x) + w(x \to y) \leq w(P'_{vx}) + w(x \to y) \leq w(P'_{vy})$

Como **no hay aristas con peso negativo**, se cumple que $w(P'_{vy}) \leq w(P')$.

Entonces $w(P) \leq w(P')$ para todo camino P' desde v a u y $d(v,u)$ es la longitud de un camino mínimo desde v a u.

- Dado un digrafo $G = (V,E)$ con *pesos positivo* en las aristas y $v \in V$, el algoritmo de **Dijkstra** determina el camino mínimo entre el vértice v y el resto de los vértices.

Demostración:

En cada iteración un nuevo vértice es incorporado a S y el algoritmo termina cuando $S = V$. Aplicando el lema anterior al finalizar la última iteración del ciclo, el algoritmo de **Dijkstra** determina, siguiendo hacia atrás $pred$ hasta llegar a v, un camino mínimo entre el vértice v y cada vértice u de V, con longitud $d(v,u)$

Bellman-Ford

```
void bellman_ford(nodo v) {
    dist[v] = 0
    i = 0
    Para u ≠ v en V:
        dist[u] = INF
    Mientras hay cambios en dist:
        i++
        dist_2 = dist
        Para u ≠ v en V:
            e = min(aristas que llegan a u)
            dist[u] = min(dist_2[u], min(dist[e.w] + w(e)))
    // Si no termina en i = n-1 tenemos ciclo negativo
    Si i = n:
        retornar "Ciclo negativo!"
}
```

Complejidad

- $O(V \cdot E)$

Correctitud

- Dado un digrafo $G = (V, E)$ si no tiene circuitos de peso negativo al finalizar la iteración k el algoritmo de **Bellman-Ford** determina los caminos mínimos de a lo sumo k aristas entre el vértice v y los demás vértices.

Demostración: Inducción en las iteraciones, k.

Caso base: Antes de ingresar por primera vez al ciclo, $k = 0$, el algoritmo fija $d(v, v) = 0$ y $d(v, u) = \infty$ para todo $u \in V \setminus \{v\}$. Esto es correcto ya que el único camino posible con 0 aristas es de v a v.

Paso inductivo: Consideramos la iteración $k \geq 1$ del algoritmo

La **HI** es: Al terminar la iteración $k'(k' < k)$ el algoritmo determina en $d^{k'}$ las longitudes de los caminos mínimos desde v al resto de los vértices con a lo sumo k' aristas.

Sea u un vértice tal que el camino mínimo desde v a u con menos cantidad de aristas contiene k aristas y P uno de estos caminos. Sea w el predecesor de u en P. Por optimalidad de subcaminos P_{vw} es un camino mínimo desde v a w con $k - 1$ aristas. Por **HI** este camino es correctamente identificado al terminar la iteración $k - 1$ del algoritmo.

Luego, en la iteración k, $d(v, u)$ recibe el valor correcto cuando es examinado la arista $w \to u$. Como este valor no puede ser mejorado (porque implicaría que P no es camino mínimo), no será modificado en las iteraciones sucesivas del algoritmo. Luego, el algoritmo calcula correctamente los camino mínimos desde v hasta al resto de los vértices.

Floyd

```
void floyd() {
    dist = L
    Para k = 1 hasta n:
        Para i = 1 hasta n:
            Para j = 1 hasta n:
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
}
```

Complejidad

- $O(V^3)$

Correctitud

- Al finalizar la iteración k del algoritmo de **Floyd** $d(i,j)$ es las longitud de los caminos mínimos desde v_i a v_j cuyos vértices intermedios son elementos de $\{v_1, \ldots, v_k\}$

Demostración: Inducción en las iteraciones, k.

Caso base: Antes de entrar por primera vez al ciclo $k = 0$ y $d = L$. Esto es correcto ya que son longitudes de los camino mínimos sin vértices intermedios, es decir, caminos que sólo son una arista. Donde L es:

$$l_{ij} = \begin{cases} 0 & si \quad i = j \\ w(v_i \to v_j) & si \quad v_i \to v_j \in E \\ \infty & si \quad v_i \to v_j \notin E \end{cases}$$

Paso inductivo: Consideramos la iteración $k \geq 1$ del algoritmo.

La hipótesis inductiva es: Al terminar la iteración $k' (k' < k)$, $d^{k'}$ tiene las longitudes de los caminos mínimos entre todos los pares de vértices cuyos vértices intermedios son elementos de $\{v_1, \ldots, v_{k'}\}$.

Dados dos vértices v_i y v_j , el camino mínimo de v_i a v_j cuyos vértices intermedios están en $\{v_1, \ldots, v_k\}$, P, tiene a v_k o no tiene a v_k. Analizando un camino mínimo, $P^1_{v_i v_j}$ que pasa por v_k y cuyos vértices intermedios son elementos del conjunto $\{v_1, \ldots, v_k\}$ y un camino mínimo $P^2_{v_i v_j}$ con vértices intermedios en $\{v_1, \ldots, v_{k-1}\}$.

$P^1_{v_i v_j} = P^1_{v_i v_k} + P^1_{v_k v_j}$, sabemos que ambos subcaminos son caminos mínimos. Por **HI** $P^1_{v_i v_k}$ tiene longitud $d^{k-1}(i,k)$ y $P^1_{v_k v_j}$ longitud $d^{k-1}(k,j)$. Por lo tanto $w(P^1_{v_i v_j}) = d^{k-1}(i,k) + d^{k-1}(k,j)$. Por **HI** $w(P^2_{v_i v_j}) = d^{k-1}(i,j)$.

P es el más corto entre P^1 y P^2. Por lo tanto $w(P) = min(w(P^1), w(P^2)) = min(d^{k-1}(i,k) + d^{k-1}(k,j), d^{k-1}(i,j))$, que es el caluclo que realiza el algoritmo. Por lo tanto es correcto.

Dantzig

```
void dantzig() {
    dist = L
    Para k = 2 hasta n:
        Para i = 1 hasta k-1:
            dist[i][k] = min(1 <= j <= k-1, dist[i][j] + dist[j][k])
            dist[k][i] = min(1 <= j <= k-1, dist[k][j] + dist[j][i])
        t = min(1 <= i <= k-1, dist[k][i] + dist[i][k])
        Si t < 0:
            retronar "Ciclo negativo!"
        Para i = 1 hasta k-1:
            Para j = 1 hasta k-1:
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])
}
```

Complejidad

- $O(V^3)$

Flujo

Definiciones

Flujo factible:

1. Para toda $e \in E_N$: $0 \leq f(e) \leq c(e)$
2. Conservación, para todo $v \in V_N$: $\sum_{e \in In(v)} = \sum_{e \in Out(v)}$

Corte:

Es el subconjunto $S \subseteq V \setminus \{t\}$ tal que $s \in S$ donde

Notación: $S, T \subseteq V$ llamamos $ST = \{(u \to v) \in E_N : u \in S, v \in T\}$

$$F = \sum_{e \in S\bar{S}} f(e) - \sum_{e \in \bar{S}S} f(e)$$

donde $\bar{S} = V \setminus S$

Corte mínimo:

$c(S) = min\{c(\bar{S} | \bar{S}\ es\ un\ corte\ de\ N)\}$

Además $F_{max} = c(S)$

Red residual:

$R(N, f) = (V, E_R)$ donde $\forall (v \to w) \in E_N$:

- $(v \to w) \in E_R \iff f((v \to w)) < c((v \to w))$
- $(w \to v) \in E_R \iff f((v \to w)) > 0$

Camino de Aumento:

- Llamo frontera a las aristas que conectan un nodo de un conjunto a otro

```
caminoAumento(N, f, R){
    S = {s}
    Mientras t no este en S and Exsita un e=(v,w) en la frontera:
        // w no está en el conjunto de v
        ant[w] = v
        S.agregar(w)
    Si t no está en S:
        retornar S corte de N
    Si no
        reconstruir P entre s y t usando ant a partir de t
        retornar P camino de aumento
}
```

Delta:

Para cada $e = (v \to w) \in P$ camino de aumento:

- $\Delta(v \to w) = \begin{cases} c(v \to w) - f(v \to w) & si\ (v \to w) \in X \\ f(v \to w) & si\ (w \to v) \in X \end{cases}$
- $\Delta(P) = min_{e \in P}\{\Delta(e)\}$

Otros:

- Si las capacidades son enteras entonces el flujo máximo también lo es

Demostración:

Caso base: Como el algoritmo comienza con un flujo inicial 0, $f_0(e) \in \mathbb{Z}_{\geq 0} \forall e \in E_N$

Paso inductivo:

HI: Para $k' < k$, $f_{k'}(e) \in \mathbb{Z}_{\geq 0} \forall e \in E_N$, donde $f_{k'}$ es la función de flujo al finalizar la iteración k' del $F\&F$

Como las capacidades son entera, por **HI**, $f_{k-1}(e) \in \mathbb{Z}_{\geq 0} \forall e \in E_N$, los valores de $\Delta(e)$ serán enteros en $R(N, f) = (V, E_R)\ \forall e \in E_R$. Luego $\Delta(P)$ es entero, y entonces $f_k(e) \in \mathbb{Z}_{\geq 0} \forall e \in E$

- Si las capacidades son racionales se pueden escalar convenientemente para transformarlas en entero.

Algoritmos

```
FF() {
    Mientras exista P camino de aumento en R(N,f):
        Para cada arista vw de P:
            Si vw esta en E
                f(vw) = f(vw) + delta(P)
            Si no
                f(wv) = f(wv) - delta(P)
}
```

Complejidad:

- $F\&F$ $F\&FO(nmU)$, donde $U = F_{max} = c(S)$
- $E\&K$ tiene $O(nm^2)$ ya que usa BFS para encontrar el camino de aumento.

Complejidad

Máquina de Turing Deterministica (DTM)

- $\sum$ el alfabeto (finito)
- Q el conjunto de estados (finito)
- $q_0 \in Q$ estado inicial
- $Q_f \subseteq Q$, estados finales (q_{si} y q_{no} para problemas de decisión)

Sobre la cinta tengo escrito el input de símbolos de $\sum$ a partir de la celda 1, y el resto de las celdas tiene blancos ($*$).

$(q_i, s_h, q_j, s_k, +1)$ Signfica que si estando en el estado q_i la cabeza lee s_h, entonces escribe s_k, se mueve a la derecha $(+1)$ y pasa al estado q_j

Una máquina M resuelve el prblema Π si para toda instancia empieza, termina y contesta bien (es decir, termina en el estado final correcto)

Complejidad: Dada por la cantidad de movimientos de la cabeza desde el estado inical hasta alcanzar un estado final.

$$T_M(n) = máx\{m : \exists x \in D_\Pi, |x| = n \wedge M\ con\ input\ x\ tarda\ m\}$$

La clase P

Un problema $\Pi \in P$ si existe una **DTM** de complejidad polinomial que lo resuelve

$$P = \{\Pi : \exists M\ DTM : M\ resuelve\ \Pi \wedge T_M(n) \in O(p(n))\ para\ algún\ polinomio\ p\}$$

Máquina de Turing NO Deterministica (NDTM)

No se pide unicidad de la quíntupla que comienza con cualquier par. En caso de que hubiera más de una quíntupla, la máquina se replica continuando cada una por una rama distinta.

Una **NDTM** resuelve el problema Π si para toda instancia de Y_Π existe una rama que llega a un estado final q_{si} y para toda instancia en $D_\Pi \setminus Y_\Pi$ ninguna rama llega a un estado final q_{si}.

Es **polinomial** para Π cuando existe una función polinomial $T(n)$ de manera que para toda instancia de Y_Π de tamaño n, alguna de las ramas termina en estado q_{si} en a lo sumo $T(n)$ pasos.

La clase NP

Un problema $\Pi \in NP$ si existe una **NDTM** polinomial que resuelve Π

NP-Completitud

- $P \subseteq NP$
- $P \neq NP$

Reducción polinomial: Sean Π_1 y Π_2 problemas de decisión. Decimo que $f : D_{\Pi_2} \to D_{\Pi_1}$ es una reducción polinomial de Π_2 en Π_1 si f se computa en tiempo polinomial y

$\forall d \in Y_{\Pi_2} \iff f(d) \in Y_{\Pi_1}$. Notación: $\Pi_2 \leq_p \Pi_1$.

Un problema $\Pi \in NP - Completo$ si:

1. $\Pi \in NP$
2. $\Pi \in NP - Hard$

Para que un problema Π sea $NP - Hard$ debe suceder que $\forall \Pi' \in NP : \Pi' \leq_p \Pi$

Teorema de Cook

Considera un problema genérico $\Pi \in NP$ y una instancia genérica $d \in D_\Pi$. A partir de la hipotética **NDTM** que resuelve Π, genera en tiempo polinomial una fórmula lógica $\varphi_{\Pi,d}$ en forma normal tal que $d \in Y_\Pi \iff \varphi_{\Pi,d}$ es satisfactible.

A partir de esto la técnica **standard** para probar que un problema $\Pi \in NP - Completo$ consta en:

1. Mostrar que $\Pi \in NP$
2. Elegir un problema Π' apropiado que se sepa que es $NP - Completo$
3. Construir una reducción polinomial f de Π' en Π

La clase Co-NP

Un problema de decisión pertenece a la clase de **Co-NP** si dada una instancia de **NO** y evidencia de la misma, puede ser verificada en tiempo polinomial.

La clase **Co-NP** es la clase de los problemas complemento de los problemas de la clase **NP**.

Demostración SAT

Sabemos que $SAT \in NP$, con lo cual solamente queda ver que $\Pi \leq_p SAT$ para todo $\Pi \in NP$. Sea $\Pi \in NP$, con lo cual existe una **NDTM** que lo resuelve.

- $\Gamma = \sum \cup \{*\}$ donde $\sum$ es el alfabeto de esta **NDTM**
- Q a su conjunto de estados
- $s \in Q$ al estado inicial
- $F \subseteq Q$ al conjunto de estados finales
- $\Delta \subseteq Q \times \Gamma \times Q \times \Gamma \times M$ al conjunto de instrucciones con $M = \{-1, 0, 1\}$

Reducimos Π a SAT del siguiente modo. Dado un input I de Π, construimos una fórmula proposicional f tal que f es satisfactible si y sólo si $I \in Y_\Pi$. Sea $p(n)$ la función polinomial de complejidad de la **NDTM**. La fórmula f contiene las siguientes proposiciones:

- T_{ijk}: La celda i contiene el símbolo j en el paso k de la ejecución de la **NDTM**
- H_{ik}: El cabezal de la **NDTM** está ubicado sobre la celda i en el paso k
- Q_{qk}: La **NDTM** está en estado q en el paso k

para $i = -p(n), \ldots, p(n),\ k = 0, \ldots, p(n),\ j \in \Gamma,\ q \in Q$

Llamamos $I = (j_0, \ldots, j_{n-1})$ y suponemos que el input comienza en la celda 0.

- T_{ij_i0}: La celda i contiene el valor j_i en tiempo 0, para $i = 0, \ldots, n-1$
- T_{i*0}: La celda i contiene el valor $*$ en tiempo 0, para $i \in \{-p(n), \ldots, p(n)\} \setminus \{0, \ldots, n-1\}$.
- H_{00}: El cabezal comienza en la celda 0.
- Q_{s0}: La máquima comienza en el estado s.
- $\neg(T_{ijk} \wedge T_{ij'k})$: La celda i contiene a lo sumo un símbolo en el paso $k (j \neq j')$
- $\vee_{j \in \Gamma} T_{ijk}$: La celda i contiene al menos un símbolo en el paso k
- $T_{ijk} \wedge T_{ij'k+1} \rightarrow H_{ik}$: Las celdas no apuntadas por el cabezal no cambian $j \neq j',\ k < p(n)$
- $\neg(Q_{qk} \wedge Q_{q'k})$: La máquina está en a lo sumo un estado en el paso k $(q \neq q')$
- $\neg(H_{ik} \wedge H_{i'k})$: El cabezal apunta a lo sumo a una celda en el paso k $(i \neq i')$
- $(H_{ik} \wedge Q_{qk} \wedge T_{i\sigma k}) \rightarrow \vee_{(q,\sigma,q',\sigma',m) \in \Delta} (H_{i+m,k+1} \wedge Q_{q',k+1} \wedge T_{i,\sigma',k+1})$: Transiciones posibles en el paso $k < p(n)$
- $\vee_{k=0}^{p(n)} \vee_{f \in F} Q_{fk}$: La máquina termina en un estado final.

Si hay un cómputo de **NDTM** con el input I que termina en un estado de F, entonces f es satisfactible asignando a las proposiciones su interpretación. Recíprocamente, si f es satisfactible entonces existe un cómputo de la **NDTM** a partir de I, siguiendo los pasos especificados por las proposiciones verdaderas.

Finalmente, la fórmula tiene $O(p(n)^2)$ proposiciones y $O(p(n)^3)$ cláusulas, con lo cual la transformación es polinomial.

Como Π es un problema arbitrario en NP, concluimos que cualuqier problema de NP reduce a SAT que, por lo tanto, es $NP - Completo$