Algoritmos de flujo máximo en grafos

–

TABLA DE CONTENIDOS

01

Algoritmo de Dinic

El problema del flujo máximo

El problema del Flujo Máximo consiste:

- Dado un grafo dirigido con pesos, G = (V, A,W), que representa las capacidades máximas de los canales, un nodo de inicio S y otro de fin T en V , **se trata de encontrar la cantidad máxima de flujo que puede circular desde S hasta T.**

- Las aristas representan canales por los que puede circular cierta cosa: transmisión de datos, redes de corriente eléctrica, líneas de oleoductos, agua, automóviles, etc.

- Los pesos de las aristas representan la capacidad máxima de un canal: velocidad de una conexión, cantidad máxima de tráfico, voltaje de una línea eléctrica, volumen máximo de agua, etc

Conceptos Básicos:

- Flujo: Circulación de unidades homogéneas de un lugar a otro.
- Capacidad de flujo: Capacidad de unidades que pueden entrar por el nodo fuente y salir por el nodo destino.
- Origen o fuente de flujo: Nodo por el que ingresa el flujo.
- Destino o Sumidero de flujo: Nodo por que sale el flujo.
- Capacidades residuales: Capacidades restantes una vez que el flujo pasa el arco.
- Camino de nivel creciente: es un camino simple en el gráfico residual en el que cada nodo tiene un nivel menor que el siguiente y todas las aristas tienen una capacidad residual positiva

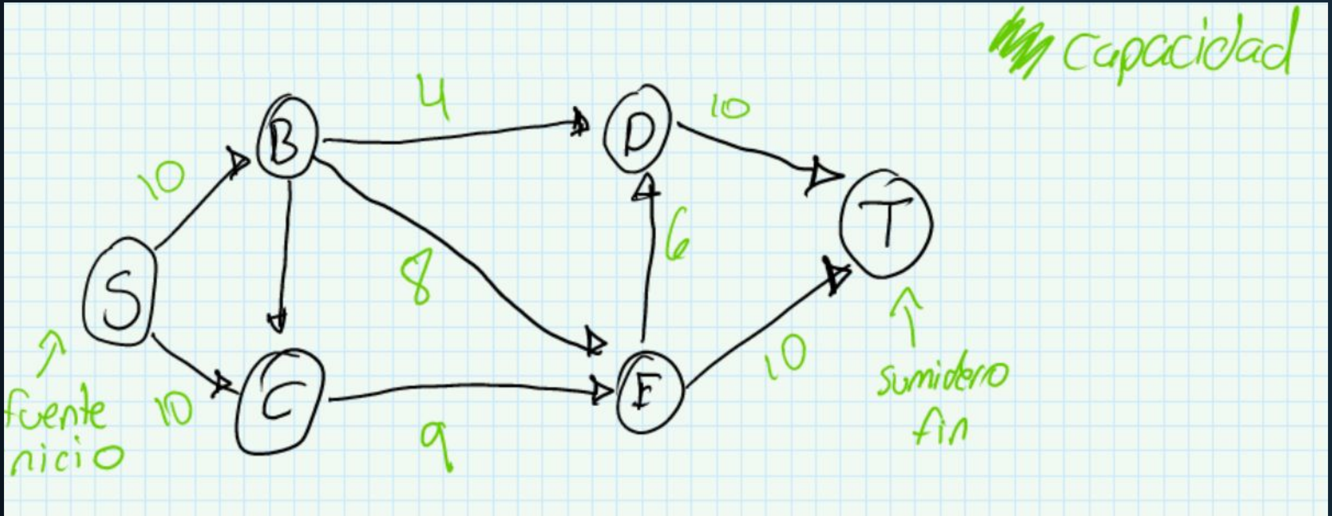
Pseudocódigo

```
Flujo_max=0
Para cada E(u,v) en G
        flow(u,v)=w(u,v)
Mientras  camino de nivel creciente de S->T
        Asignar niveles a los vertices
        Mientras sea posible un mayor flujo desde s hasta t
                min_cap=minima capacidad residual en el camino
                Flujo_max=Flujo_max+min_cap
                Para cada arista en el camino
                        flow(u,v)= flow(u,v)-min_cap
                        flow(v,u)= flow(v,u)+min_cap
Retorna flujo_max
```

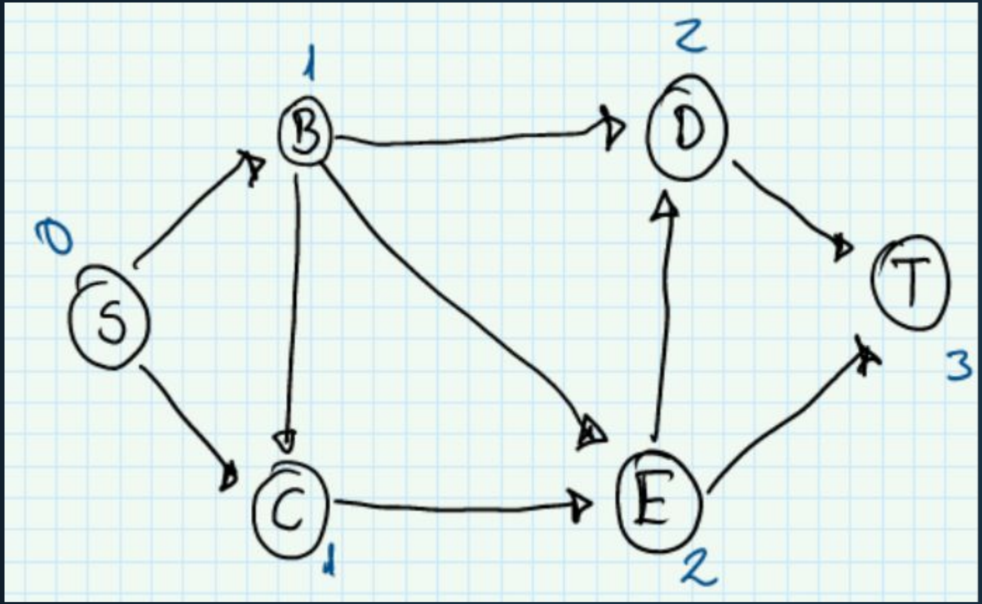
Algoritmo de Dinic

Problema
Una empresa petrolera quiere enviar la mayor cantidad de petróleo del punto S al punto T, el sistema de tuberías estará representado por un grafo y el peso de las aristas representarán la capacidad de las tuberías

Algoritmo de dinic



Aplicando Bfs

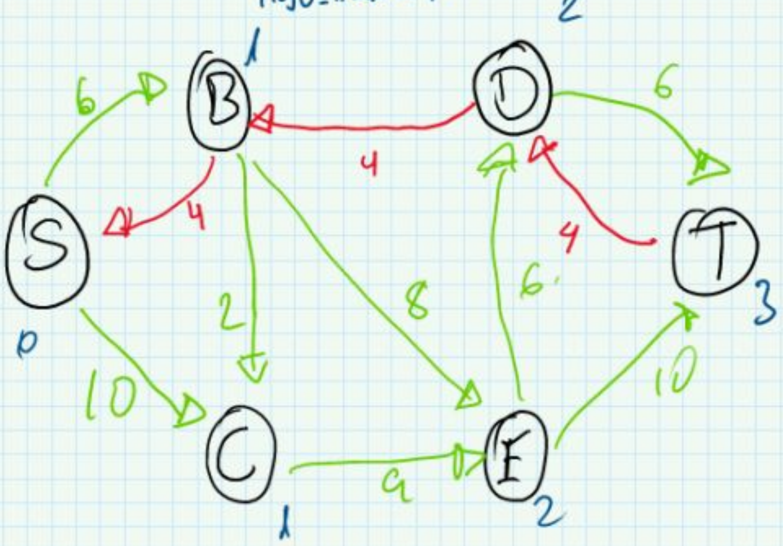


$S \rightarrow B \rightarrow D \rightarrow T$

10 4 10

min_cap=4

flujo_max = 4



$S \to B \to C \to E \to T$

0 1 1 2 3 ✗

no aumenta

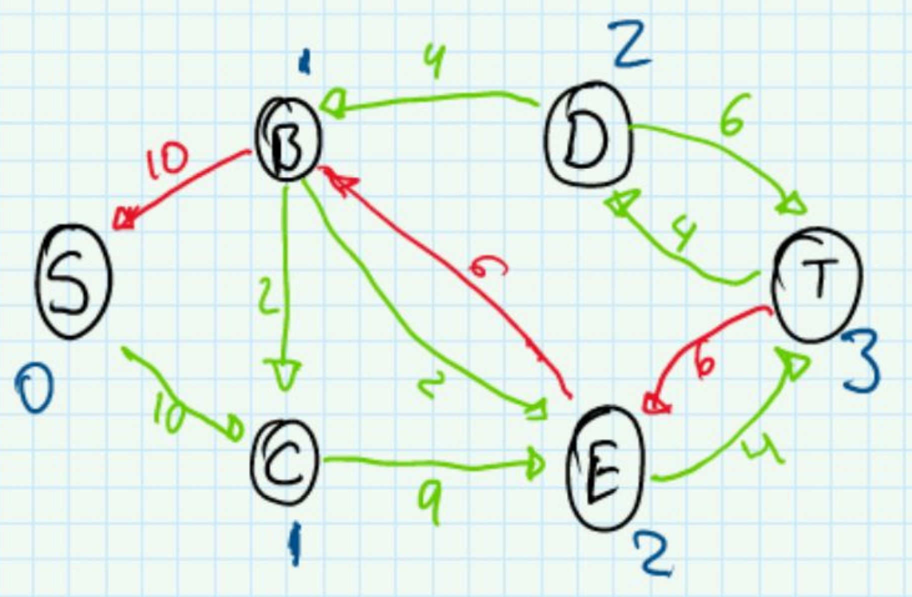
$S \to B \to E \to T$ ✓

0 1 2 3

6 8 10

min_cap = 6

flujo_max = 10



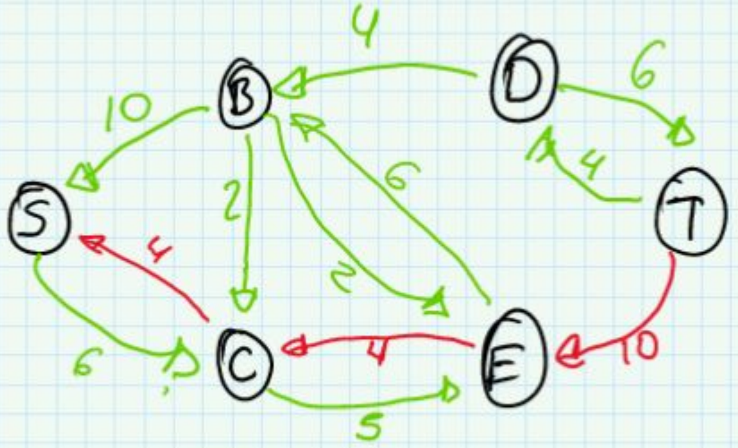
$S \to C \to E \to T$ ✓

0 1 2 3

10 9 4

min_cap = 4

flujo_max = 14



$S \to$	$C \to$	$E \to$	$D \to$	T
0	1	2	2	3

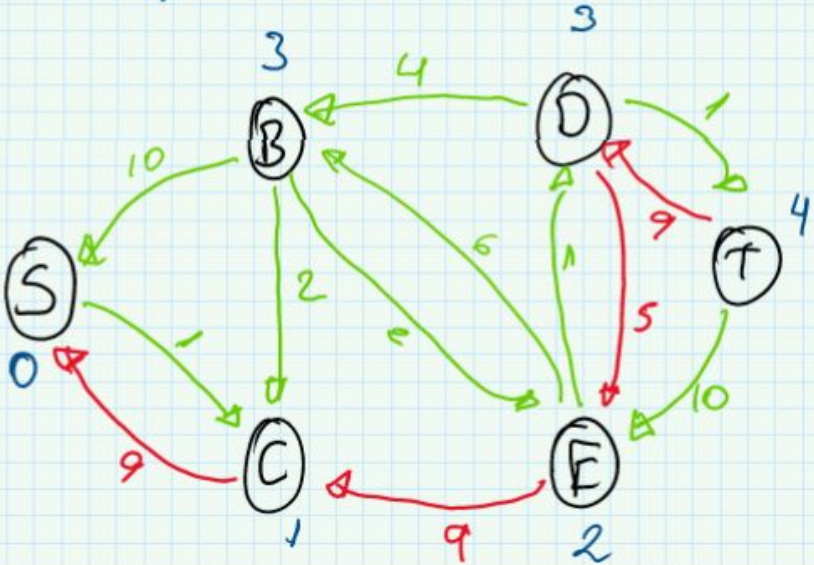
0 1 2 3 4

$S \to C \to E \to D \to T$

6 5 6 6

$min\_cap = 5$

$Flujo\_max = 19$



02

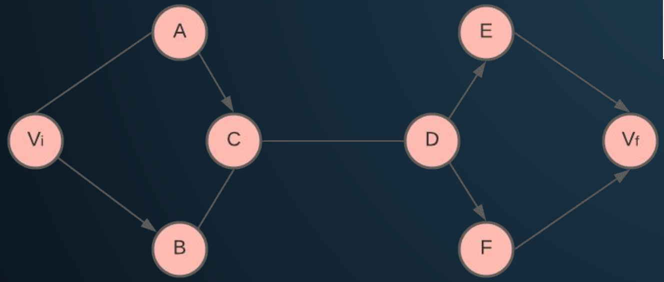
Algoritmo de Ford-Fulkerson De Edmonds Karp 1972

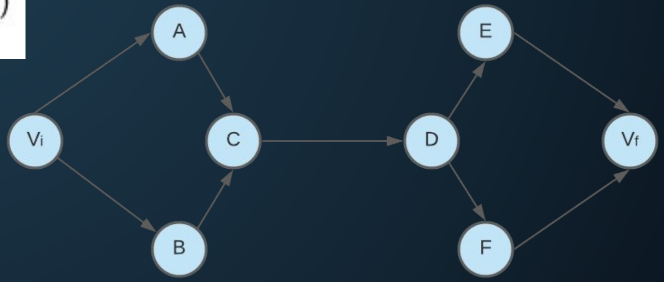
02.1 Problema de flujo máximo

Problema de flujo máximo

- Estando en una Red = Grafo dirigido conexo , con pesos en las aristas
- Tener una única fuente Vi
- Tener un único sumidero Vf
- No tener aristas no dirigidas
- Los vértices no tendrán capacidad
- El valor del flujo es el total que sale de la fuente , que es igual al total de flujo que llega al sumidero

$$|f| = \sum_{v \in V} f(s, v)$$





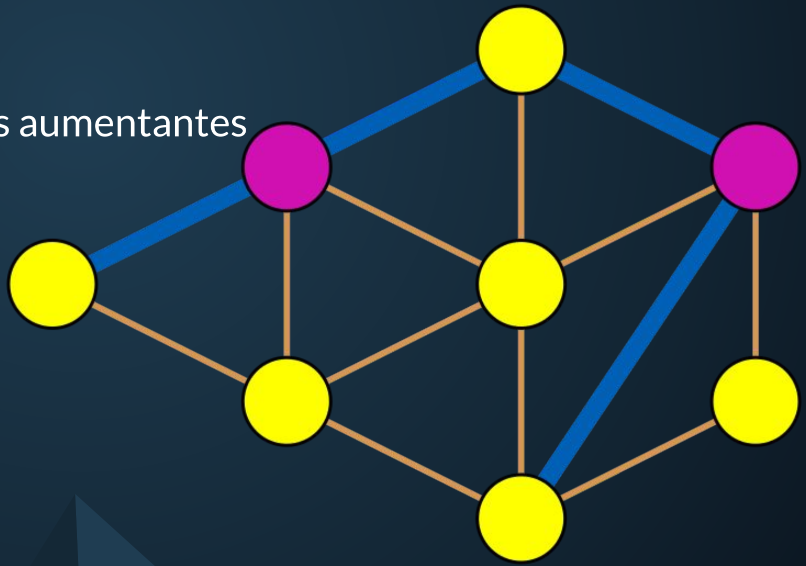
02.2

Conceptos

Concepto

El algoritmo :

- Caminos en que se pueda aumentar el flujo
- **Ford-Fulkerson**
- Llegar al Flujo máximo
- **Edmonds - Karp**
- Orden para la búsqueda de caminos aumentantes
- Camino -> óptimo (corto)
- Aplicar un **DFS** (wa < w[ ])

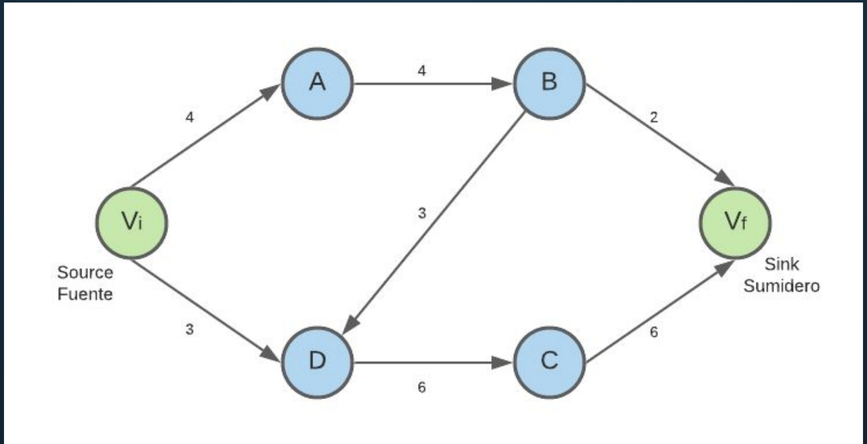


02.3

Problema

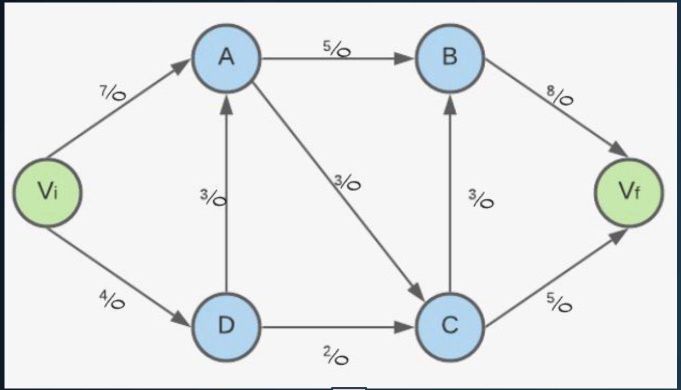
Ejercicio-Problema

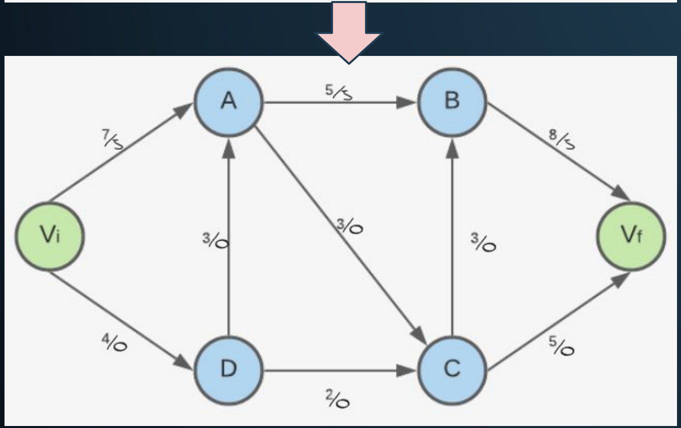
En jefe de obra , ha diseñado una red de tubería de agua en una casa , el cual es representado mediante un grafo , en el que **v** sería las aristas(tuberias de agua) , **u** vendrian ser los vertices (caños o salida de agua) y por último **w** que sería la capacidad máxima de la tubería , el objetivo es cuánta agua puede fluir o empujar efectivamente desde la entrada hasta la salida de la tubería (flujo máximo).



Método Ford-Fulkerson

Ejercicio-Problema- Ford-Fulkerson





Valor flujo = 0

Recorrido: A simple vista

Grafo auxiliar

Recorrido : Vi-A-B-Vf

Vi -> A = 7

A -> B = 5

B -> Vf = 8

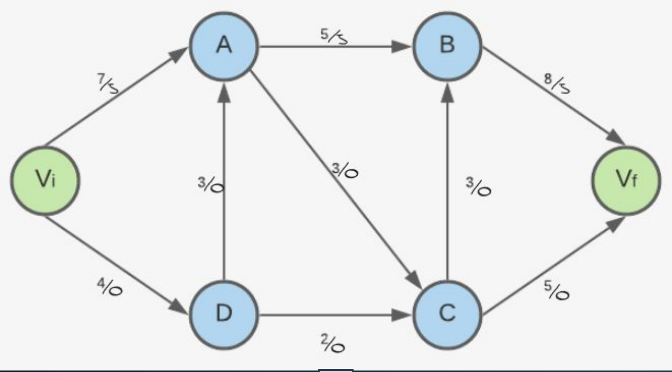
Despues :

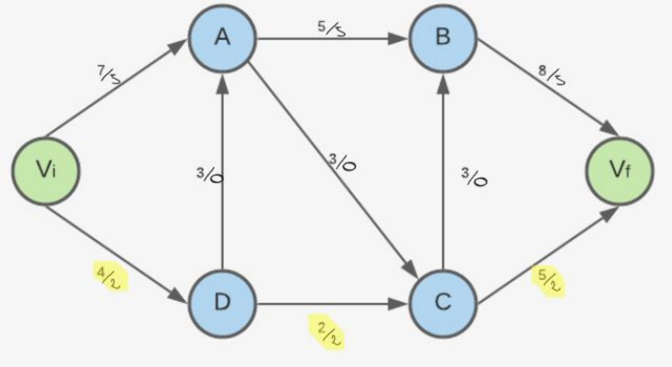
Vi -> A = 0+5 = 5

A -> B = 0+5 = 5

B -> Vf = 0+5 = 5

Ejercicio-Problema- Ford-Fulkerson





Valor flujo = 0+5=5

Recorrido: A simple vista

Grafo auxiliar

Recorrido : Vi-D-C-Vf

Vi -> D = 4

D -> C = 2

C -> Vf = 5

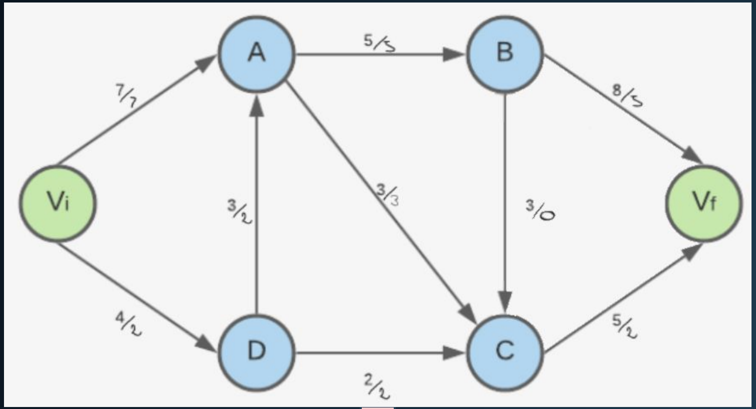
Despues :

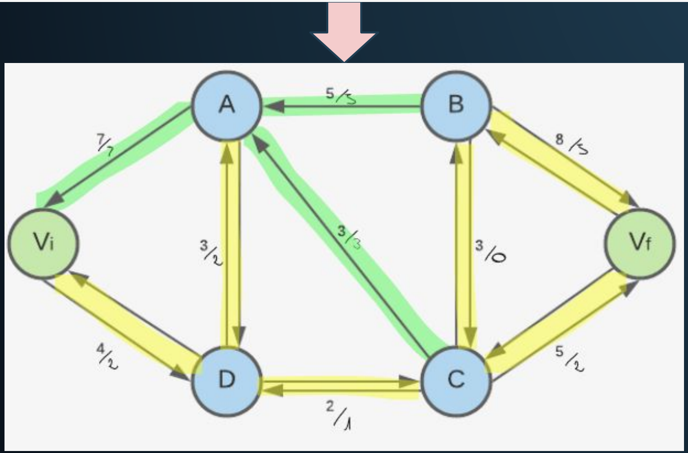
Vi -> A = 0+2 = 2

A -> B = 0+2 = 2

B -> Vf = 0+2 = 3

Ejercicio-Problema- Ford-Fulkerson- full



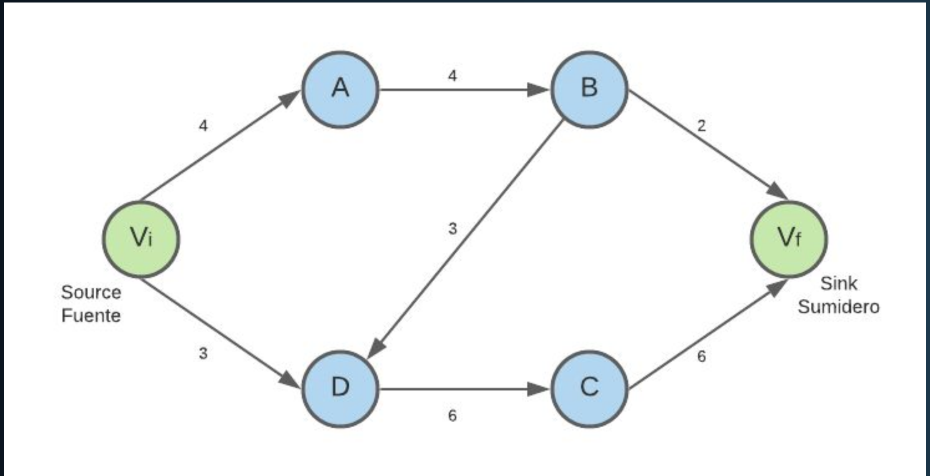


Valor flujo = 0+5+2=7
Recorrido: A simple vista
Grafo auxiliar

Recorrido : Vi-D-C-B-Vf
Vi -> D = 2
D -> C = 1
C -> B = 3
B -> Vf = 5
Despues :
Vi -> D = 2+1 = 3
D -> C = 1+1 = 2
C -> B = 2-1 = 1
B -> Vf = 5+1=6

Método De Edmonds Karp

Ejercicio-Problema- Edmonds Karp



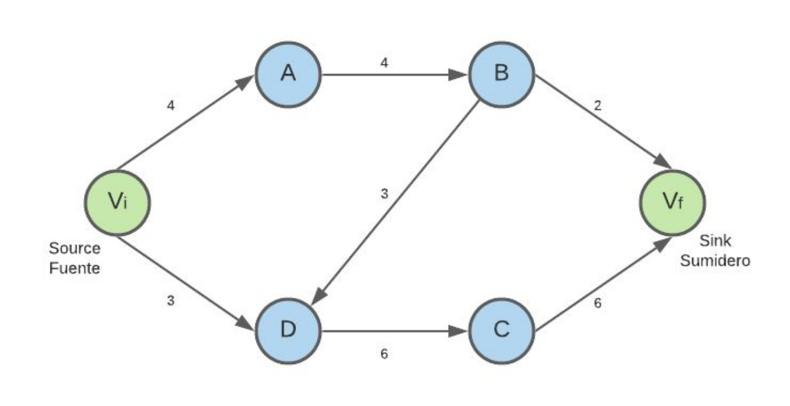
- Valor flujo (flow) <= capacidad max de la arista.
- El valor de entrada sera igual a la de la salida (excepto la fuente y sumidero).

Ejercicio-Problema- Edmonds Karp

Lógica

```
max_flow = 0
for e(u, v) in G:
    flow(u, v) = w(u,v)
while
  min_cap = min
  max_flow = max_flow + min_cap
  for e(u, v) in p:
      flow(u, v) = flow(u, v) - min_cap
      flow(v, u) = flow(v, u) + min_cap
return max_flow
```

Ejercicio-Problema- Edmonds Karp



- Max_flow = 0 + 2
- Escogemos recorrido :
  - Vi -> A -> B -> Vf
- Buscamos la minima cap

  Vi -> A = 4

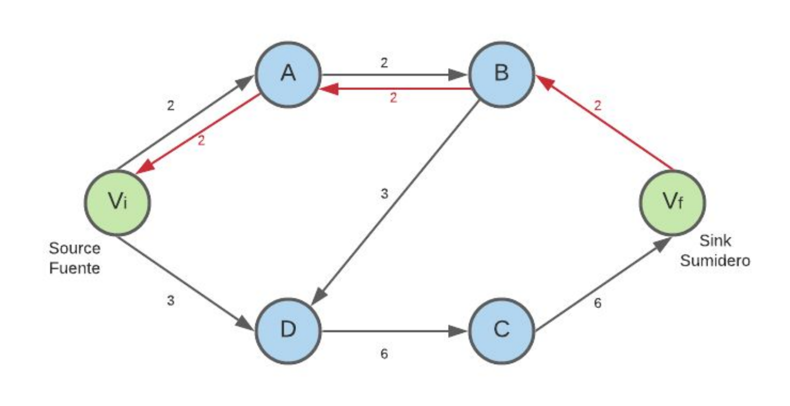
  A -> B = 4

  B -> Vf =2

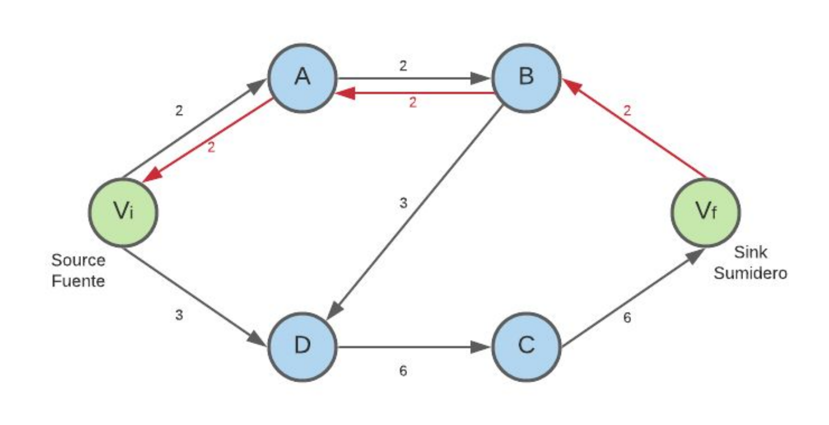
Flow(Vi,A)= 4 - 2 = 2 & (A,Vi)= 0 + 2 = 2

(A, B) = 4 - 2 = 2 & (B,A) = 0 + 2 = 2

(B,Vf) = 2 - 2 = **0** & (Vf,B)= 0 + 2 = 2



Ejercicio-Problema- Edmonds Karp

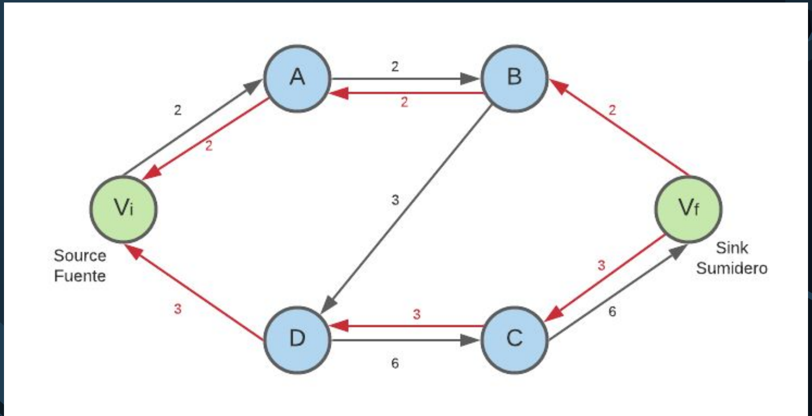


- Max_flow = 0 + 2 + 3
- Escogemos recorrido :
  - Vi -> D -> C -> Vf
- Buscamos la minima cap

  Vi -> D = 3

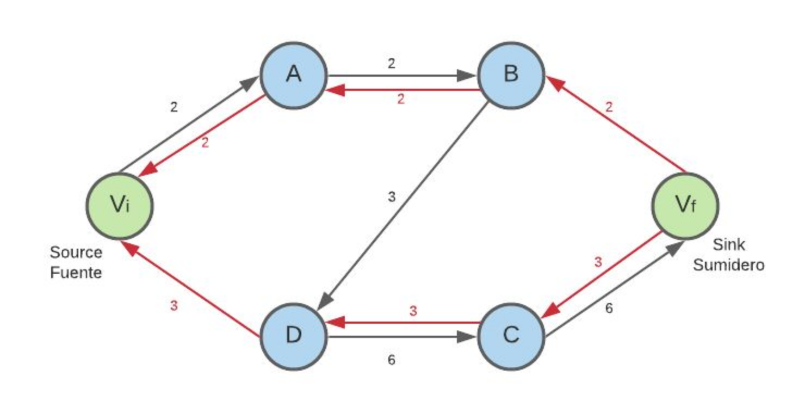
  D -> C = 6

  C -> Vf = 6



Flow(Vi,D)= 3 - 3 = 0 & (D,Vi)= 0 + 3 = 3
(D, C) = 6 - 3 = 3 & (C,D) = 0 + 3 = 3
(C,Vf) = 6 - 3 = 3& (Vf,C)= 0 + 3 = 3

Ejercicio-Problema- Edmonds Karp último



- Max_flow = 0 + 2 + 3 + 2
- Ultimo y unico camino :
  - Vi -> A -> B -> D -> C ->Vf
- Buscamos la minima cap

  Vi -> A = 2

  A -> B = 2

  B -> D = 3

  D -> C = 6

  C -> Vf = 6

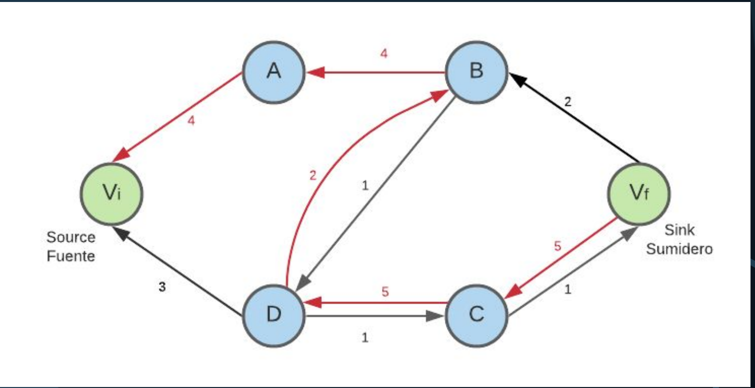
Flow(Vi,A)= 2 - 2 = 0 & (A,Vi)= 2 + 2 = 4

(A, B) = 2 - 2 = 0 & (B,A) = 2 + 2 = 4

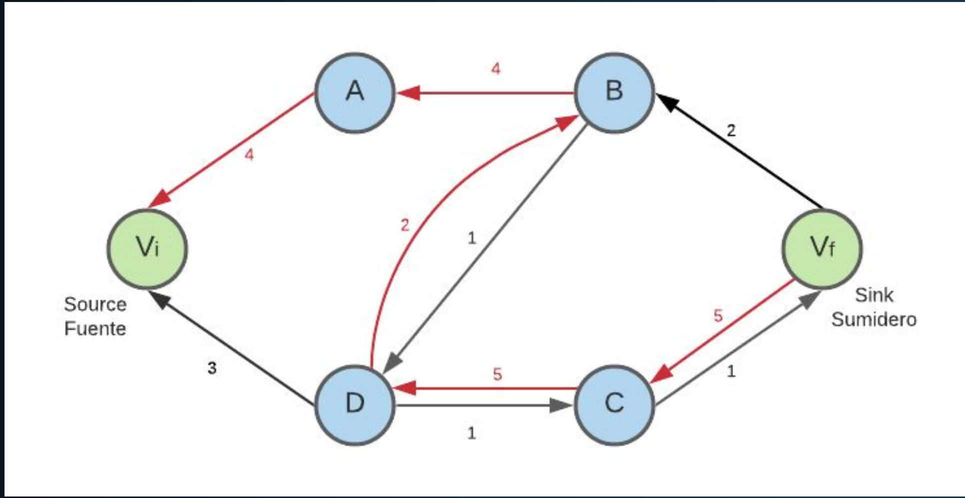
(B, D) = 3 - 2 = 1 & (D,B) = 0 + 2 = 2

(D, C) = 3 - 2 = 1 & (C,D) = 3 + 2 = 5

(C,Vf) = 3 - 2 = 1 & (Vf,C)= 3 + 2 = 5



Ejercicio-Problema- Edmonds Karp último



- Max_flow = 0 + 2 + 3 + 2 = 7
- Complejidad: O(v e^2)

v= vertices y e =aristas

```
max_flow = 0+2+3+2=7
for e(u, v) in G:
    flow(u, v) = w(u,v)
while
  min_cap = min
  max_flow = max_flow + min_cap
  for e(u, v) in p:
      flow(u, v) = flow(u, v) - min_cap
      flow(v, u) = flow(v, u) + min_cap
return max_flow
```

Ejercicio-Problema- Edmonds Karp último

```cpp
int ford_fulkerson(vector<vector<int>>& graph, int source, int sink) {
    vector<int> parent(graph.size(), -1);//-1
    vector<vector<int>> residualGraph = graph;
    int min_cap = 0, max_flow = 0;
    //cout<<" parentesco U: "<<parent[source]<<endl;
    //cout<<" parentesco V: "<<parent[sink]<<endl;
    while (min_cap = bfs(source, sink, parent, residualGraph)) {
        max_flow += min_cap;
        int u = sink;
        //cout<<" parentesco U1: "<<parent[source]<<endl;
        //cout<<" parentesco V1: "<<parent[sink]<<endl;
        while (u != source) {
            int v = parent[u];

            residualGraph[u][v] += min_cap;
            residualGraph[v][u] -= min_cap;
            u = v;
            cout<<" actualizacion del flujo max : "<<max_flow<<endl;
        }
    }
    return max_flow;
}
```

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <bits/stdc++.h>

using namespace std;

int bfs(int source, int sink, vector<int>& parent, vector<vector<int>>& residualGraph) {

    fill(parent.begin(), parent.end(), -1);
    int V = residualGraph.size();
    parent[source] = -2;
    queue<pair<int, int>> q;
    q.push({source, INT_MAX});

    while (!q.empty()) {
        int u = q.front().first;
        int capacity = q.front().second;
        q.pop();
        for (int av=0; av < V; av++) {
            if (u != av && parent[av] == -1 && residualGraph[u][av] != 0) {
                parent[av] = u;
                int min_cap = min(capacity, residualGraph[u][av]);
                if (av == sink)
                    return min_cap;
                q.push({av, min_cap});
            }
        }
    }
    return 0;


    ford_fulkerson(vector<vector<int>>& graph, int source, int sink) {
    vector<int> parent(graph.size(), -1);//-1
    vector<vector<int>> residualGraph = graph;
    int min_cap = 0, max_flow = 0;
    //cout<<" parentesco U: "<<parent[source]<<endl;
    //cout<<" parentesco V: "<<parent[sink]<<endl;
    while (min_cap = bfs(source, sink, parent, residualGraph)) {
        max_flow += min_cap;
        int u = sink;
        //cout<<" parentesco U1: "<<parent[source]<<endl;
        //cout<<" parentesco V1: "<<parent[sink]<<endl;
```

Ejercicio-Problema- Edmonds Karp último

```cpp
    //cout<<" parentesco V: "<<parent[sink]<<endl;
    while (min_cap = bfs(source, sink, parent, residualGraph)) {
        max_flow += min_cap;
        int u = sink;
        //cout<<" parentesco U1: "<<parent[source]<<endl;
        //cout<<" parentesco V1: "<<parent[sink]<<endl;
        while (u != source) {
            int v = parent[u];

            residualGraph[u][v] += min_cap;
            residualGraph[v][u] -= min_cap;
            u = v;
            cout<<" actualizacion del flujo max : "<<max_flow<<endl;
        }
    }
    return max_flow;
}

void addEdge(vector<vector<int>>& graph,int u, int v, int w)
{
    graph[u][v] = w;
}

int main()
{
    int V = 6;
    vector<vector<int>> graph(V, vector<int> (V, 0));

    addEdge(graph, 0, 1, 4);
    addEdge(graph, 0, 3, 3);

    addEdge(graph, 1, 2, 4);

    addEdge(graph, 2, 3, 3);
    addEdge(graph, 2, 5, 2);

    addEdge(graph, 3, 4, 6);

    addEdge(graph, 4, 5, 6);

    cout << "flujo maximo: " << ford_fulkerson(graph, 0, 5) << endl;
    return 0;
}
```

Ejercicio-Problema- Edmonds Karp último

C:\Program Files (x86)\Zinjal\bin\runner.exe

```
 actualizacion del flujo max : 2
 actualizacion del flujo max : 2
 actualizacion del flujo max : 2
 actualizacion del flujo max : 5
 actualizacion del flujo max : 5
 actualizacion del flujo max : 5
 actualizacion del flujo max : 7
 actualizacion del flujo max : 7
 actualizacion del flujo max : 7
 actualizacion del flujo max : 7
 actualizacion del flujo max : 7
flujo maximo: 7


<< El programa ha finalizado: codigo de salida: 0 >>
<< Presione enter para cerrar esta ventana >>
```

- Max_flow = 0 + 2 + 3 + 2 = 7
- Complejidad: $O(vE^2)$

03

Algoritmo Maximum Bipartite Matching

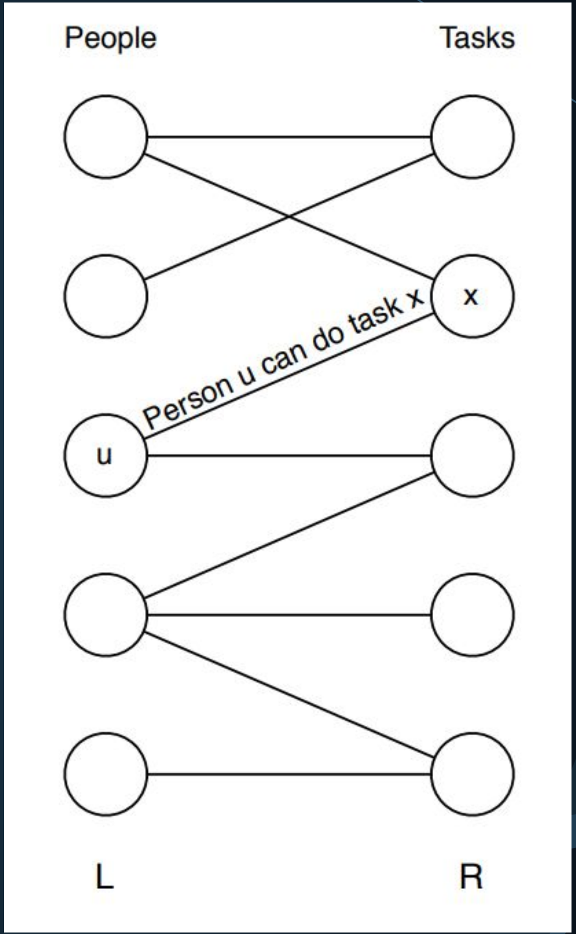
Un ‘**matching**’ en un gráfico bipartito es un conjunto de aristas elegidas de tal manera que no haya dos aristas que compartan un punto final.

Un ‘**maximum matching**’ es una coincidencia de tamaño máximo . En una coincidencia máxima, si se le agrega alguna arista, ya no es una coincidencia.

- Supongamos que tenemos un conjunto de personas L y conjunto de trabajos R.

- Cada persona puede hacer solo algunos de los trabajos.

- Esto se presenta en el siguiente grafo bipartito

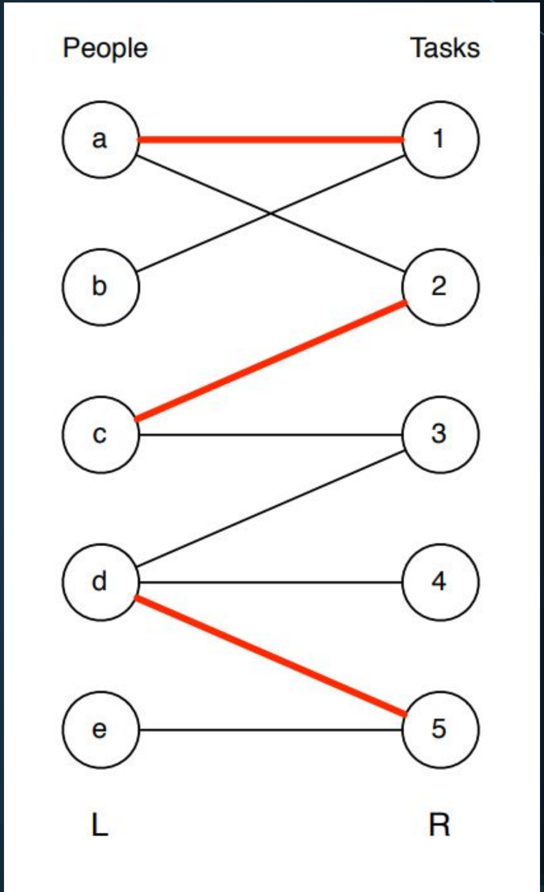


- Una coincidencia da un asignación de personas a Tareas.

- Quiere conseguir tantas tareas hecho como sea posible.

- Entonces, quiero un máximo coincidencia: uno que contiene tantos bordes como
- posible.

- (Este no es **maximum matching**).



Maximum Bipartite Matching

Dado un grafo bipartito $G = (A \cup B, E)$, encuentre un $S \subseteq A \times B$ que sea una coincidencia y es lo más grande posible.

Notas:

- Se nos da A y B para que no tengamos que encontrarlos.
- S es una coincidencia perfecta si todos los vértices coinciden.
- ‘**Maximum**’ no es lo mismo que ‘**Maximal**’: Greedy llegaría a ‘máximal’

SOLUCIÓN

- Dada una instancia de emparejamiento bipartito.

- Cree una instancia de flujo de red.

- Donde la solución al problema de flujo de red puede ser utilizado fácilmente para encontrar el solución al bipartito pareo.

Instancia de Maximum Bipartite Matching

↓ Se transforma

Instancia de Network Flow

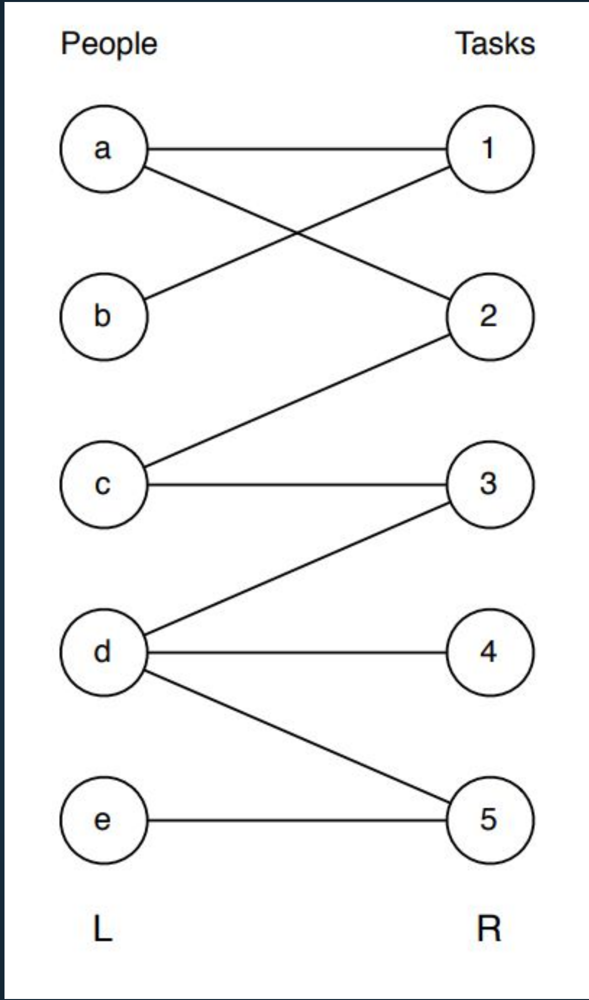
Reduciendo ‘Bipartite Matching’ a ‘Net Flow’

Usando ‘Net Flow’ para resolver ‘Bipartite Matching’

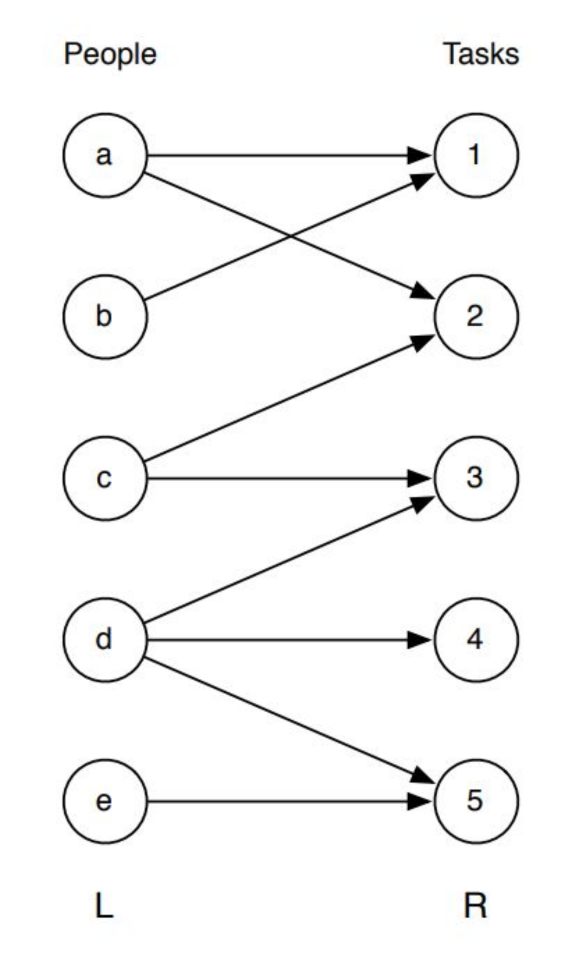
Recordando:

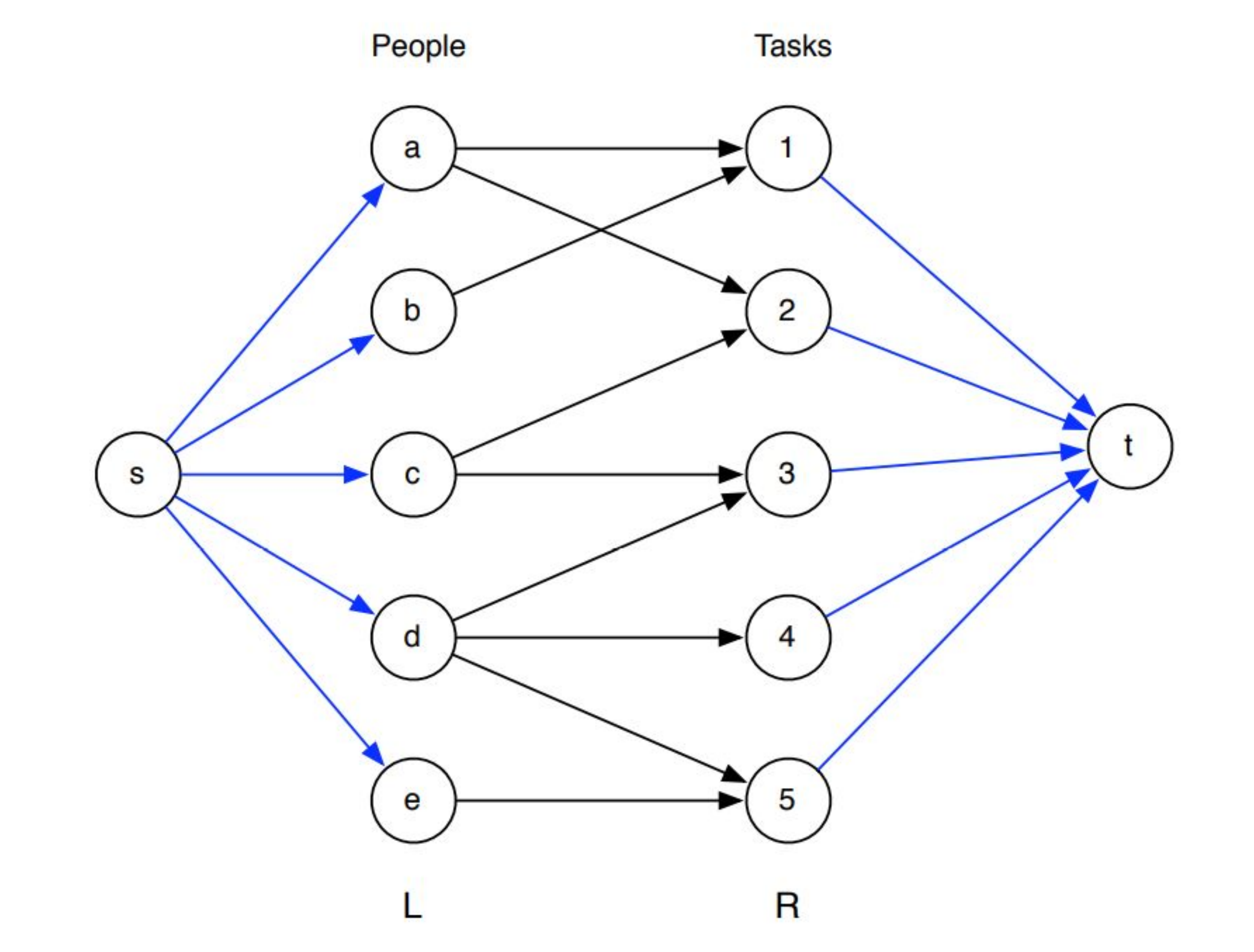
1. Dado el gráfo bipartito $G = (A \cup B, E)$, directo los bordes de A a B.
2. Agregue nuevos vértices s y t.
3. Agregue una arista desde s a cada vértice en A.
4. Agregue una arista de cada vértice de B a t.
5. Realice todas las capacidades 1.
6. Resuelva el problema de flujo de red máximo en este nuevo gráfico G’

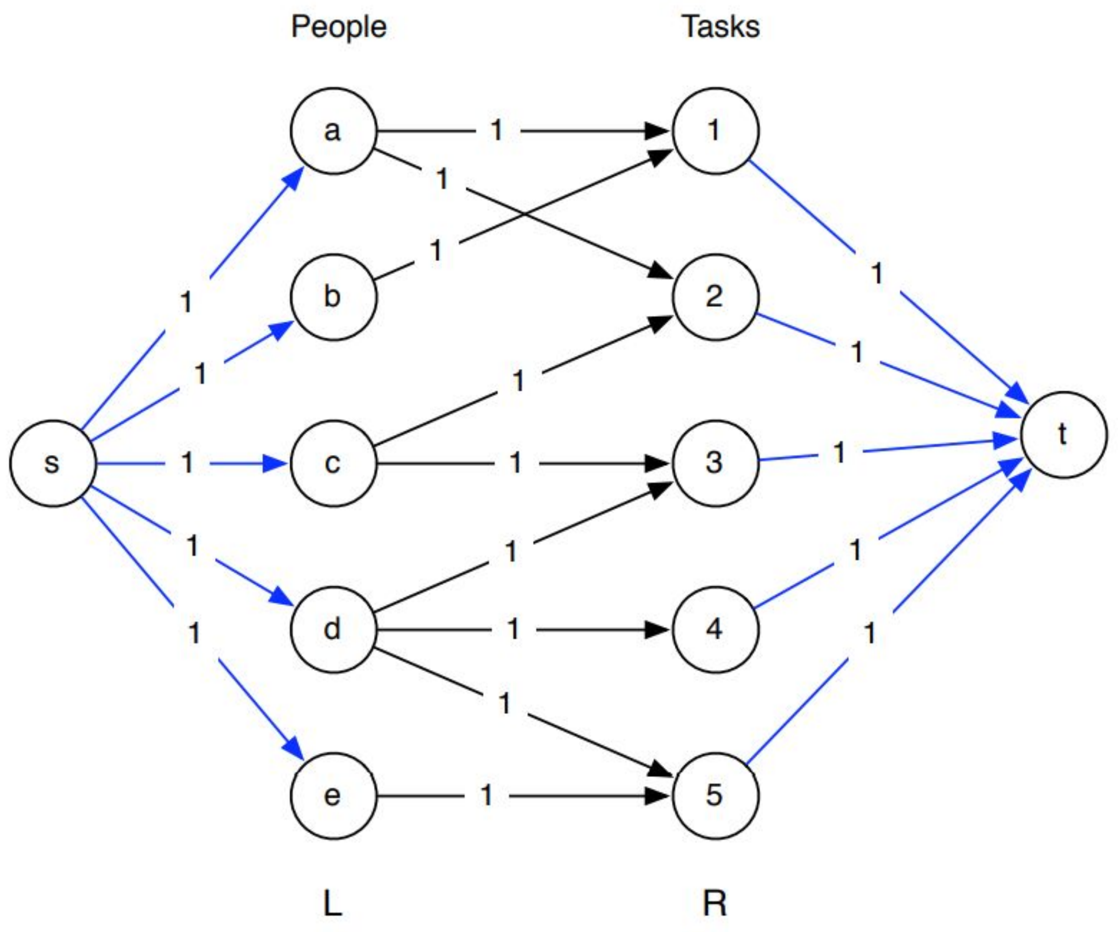
Los bordes usados en el flujo máximo de la red serán los corresponden a el ‘maximum matching’



People
Tasks
a
b
c
d
e
1
2
3
4
5
L
R


People
Tasks
a
b
c
d
e
1
2
3
4
5
L
R


People
Tasks
a
b
c
d
e
1
2
3
4
5
s
t
L
R

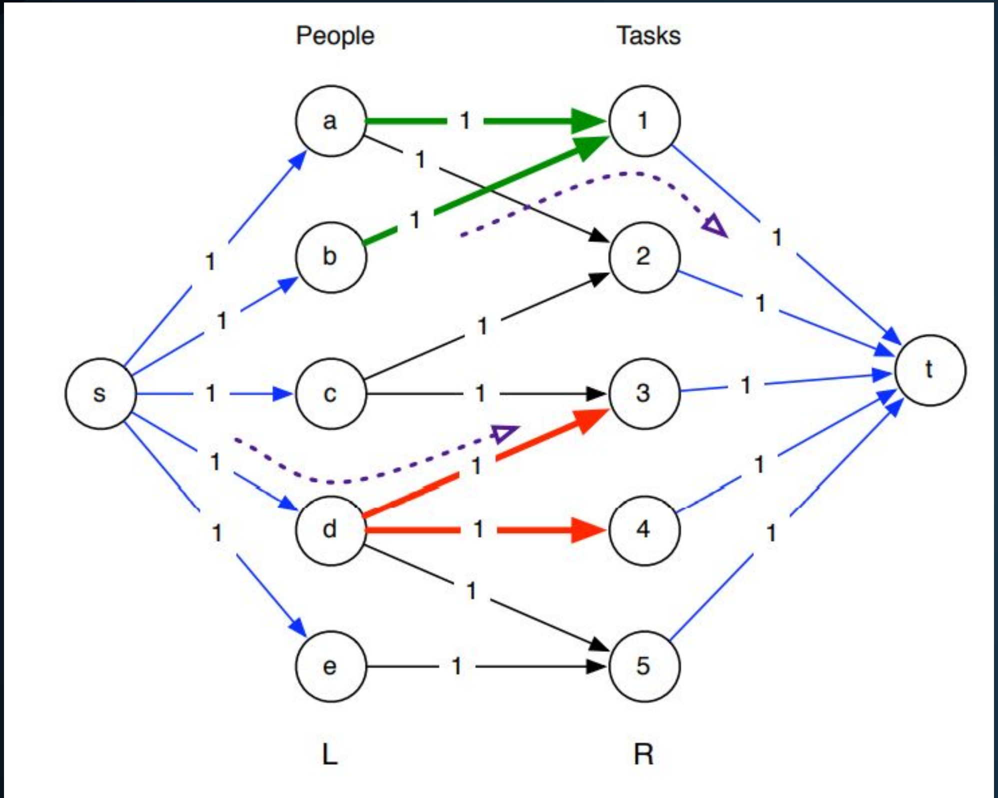

People
Tasks
a
b
c
d
e
1
2
3
4
5
s
t
L
R

Sea M el conjunto de aristas que van de A a B que usar.
Probemos que:

1. M es un ‘matching’
2. M es el ‘maximum matching’

M es un ‘matching’

Podemos elegir como máximo una arista dejando cualquier nodo en A.
Podemos elegir como máximo un borde que ingrese a cualquier nodo en B.

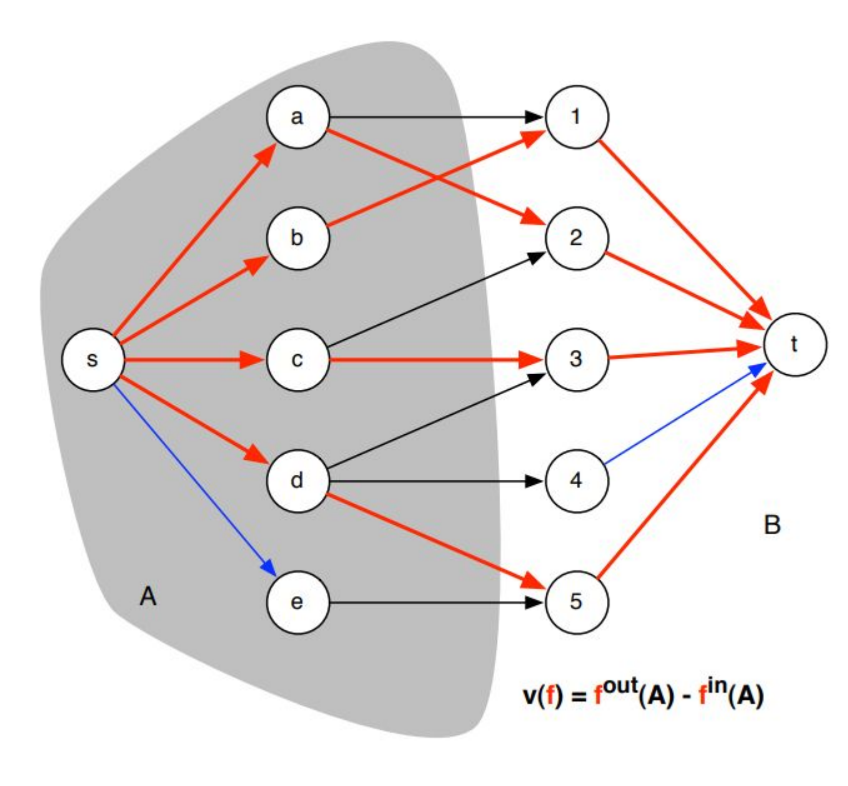


Si elegimos más de 1, no podríamos tener un flujo equilibrado.

Correspondencia entre flujos y emparejamientos

- Si hay una coincidencia de k aristas, hay un flujo f de valor k.

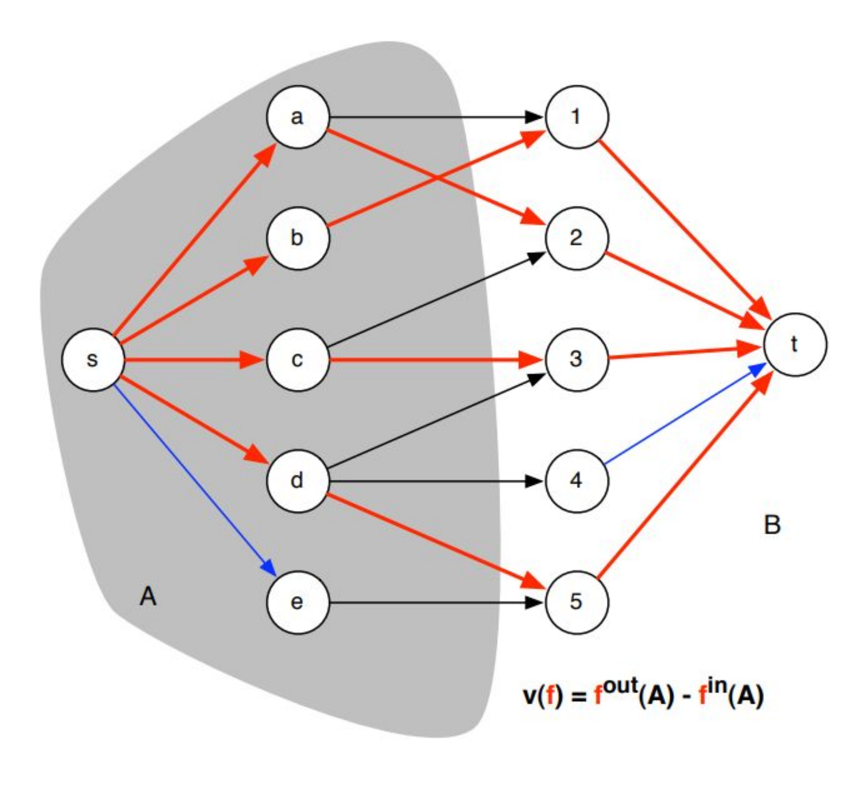
- Si hay un flujo f de valor k, hay una coincidencia con k aristas.



Correspondencia entre flujos y emparejamientos

- Si hay una coincidencia de k aristas, hay un flujo f de valor k.
  - f tiene 1 unidad de flujo a través de cada uno de los k bordes.
  - ≤ 1 unidad sale y entra en cada nodo (excepto s, t)

- Si hay un flujo f de valor k, hay una coincidencia con k aristas.



M es lo más grande posible

- Encontramos el ‘**maximum flow**’ f (digamos con k aristas).
- Esto corresponde a un **‘matching’** M de k aristas.
- Si hubiera un ‘matching’ con > k aristas, habríamos encontrado un flujo con valor> k, lo que contradice que f sea máximo.
- Por tanto, M es máximo.

Código

```
#include <iostream>
#include <string.h>
using namespace std;

// M es el numero de aplicantes y N el numero de trabajos
#define M 6
#define N 6

```

```
// Una función DFS recursiva que retorna TRUE si hay match posible para el vertice u
bool bpm(bool bpGraph[M][N], int u, bool seen[], int matchR[]){
    // Prueba todos los trabajos 1x1
    for (int v = 0; v < N; v++)    {
        //Si el aplicante u esta interesado en el trabajo v y v no fue visitado
        if (bpGraph[u][v] && !seen[v])        {
            // Marca v como visitado
            seen[v] = true;

            /*Si el trabajo 'v' no está asignado a un solicitante o
            el solicitante previamente asignado para el trabajo v
            (que es matchR [v]) tiene un trabajo alternativo disponible.
             Dado que v está marcado como visitado en la línea anterior,
             matchR [v] en la siguiente llamada recursiva no obtendrá
             el trabajo 'v' nuevamente*/

            if (matchR[v] < 0 || bpm(bpGraph, matchR[v], seen, matchR))          {
                matchR[v] = u;
                return true;
            }
        }
    }
    return false;
}
```

```cpp
// Retorna el maximo numero de matching de M a N
int maxBPM(bool bpGraph[M][N])
{
    // Una matriz para realizar un seguimiento de los solicitantes asignados a los trabajos.
    //El valor de matchR [i] es el número de solicitante asignado al trabajo i, el valor -1 indica que no se asignó a nadie.
    int matchR[N];

    // Inicialmente todos los trabajos estan disponibles
    memset(matchR, -1, sizeof(matchR));

    // Contador de trabajos asignado a los aplicantes
    int result = 0;
    for (int u = 0; u < M; u++)
    {
        // Marca todos los trabajos como no visitados para el siguiente aplicante
        bool seen[N];
        memset(seen, 0, sizeof(seen));

        // Verifica si el aplicante u puede conseguir el trabajo
        if (bpm(bpGraph, u, seen, matchR))
            result++;
    }
    return result;
}
```

```
}

int main(){
    bool bpGraph[M][N] = {{0, 1, 1, 0, 0, 0},
                          {1, 0, 0, 1, 0, 0},
                          {0, 0, 1, 0, 0, 0},
                          {0, 0, 1, 1, 0, 0},
                          {0, 0, 0, 0, 0, 0},
                          {0, 0, 0, 0, 0, 1}};

    cout << "El máximo numero de aplicantes que puede conseguir el trabajo es: "<< maxBPM(bpGraph);

    return 0;
}
```

PROBLEMAS SALIDA TERMINAL CONSOLA DE DEPURACIÓN

```
nnerFile }
El m├íximo numero de aplicantes que puede conseguir el trabajo es: 5
PS C:\Users\mauri\AppData\Local\Temp>
```

Tiempo de Ejecución

El tiempo de ejecución de Ford-Fulkerson es O (m’C) donde m’ es el número de aristas, y $C = \sum_{e \text{ leaving } s} c_e$.

- C = | A | = n
- El número de aristas en G’ es igual al número de aristas en G (m) más 2n.
- Entonces, el tiempo de ejecución es O ((m + 2n) n) = (mn + n 2) = O (mn)

CONCLUSIONES

- Fold-Fulkerson puede encontrar un ‘maximum matching’ en un grafo bipartito en O (mn).
- Hacemos esto reduciendo el problema de ‘Maximum Bipartite Matching’ con ‘Net Flow’.

THANKS!

Do you have any questions?
youremail@freepik.com
+91 620 421 838 yourcompany.com