Complejidad Computacional y Programación Dinámica

Dr. Ing. Claudio Risso (crisso@fing.edu.uy)

Instituto de Computación (FING - UDELAR)

Metaheurísticas y Optimización sobre Redes
(Agosto 24, 2020)

¿Cuál es la eficiencia de un algoritmo?

Let $f(x)$ and $g(x)$ be two real functions defined on some input string set, whose outputs (positive reals) respectively represent the time expended by two algorithms $\mathcal{A}_f$ and $\mathcal{A}_g$, running upon the same computer, to find a solution to an instance associated with each input string.

Definition

We say that g is of *higher complexity order* than f (denoted as $f(x) = O(g(x))$, iff, there is a positive constant M such that for all sufficiently large strings x, $f(x)$ is at most M multiplied by $g(x)$. That is, exists x_0 and M such that: $f(x) \leq Mg(x)$, for all $|x| > x_0$.

Coefficients and lower order terms are usually excluded. Example: if the time required on all inputs of size n is at most $5n^3 + 3n$, we say that the *asymptotic time complexity* is $O(n^3)$.

¿Qué representa en términos prácticos?

Suppose that computing power (in terms of computations per second) increases by a factor of 1000 at decade, and we have to choose between four different algorithms to find solutions for a critical application, which demands for an answer within a day. All of them can find solutions within a day for instances of size 100. Besides, it is known that complexity orders for these algorithms are: n^2, n^{10}, 10^n and $n!$ respectively.

¿Cuáles son los tamaños de los problemas que podemos resolver en un día?

¿Qué representa en términos prácticos?

Suppose that computing power (in terms of computations per second) increases by a factor of 1000 at decade, and we have to choose between four different algorithms to find solutions for a critical application, which demands for an answer within a day. All of them can find solutions within a day for instances of size 100. Besides, it is known that complexity orders for these algorithms are: n^2, n^{10}, 10^n and $n!$ respectively.

year	pod. comp.	$O(n^2)$	$O(n^{10})$	$O(10^n)$	$O(n!)$
2020	1.000	100	100	100	100
2030	1.000.000	3,162	200	103	102
2040	10^9	100,000	398	106	104
2050	10^{12}	3,162,278	794	109	105
2060	10^{15}	100,000,000	1,585	112	106

Década tras década el tamaño de una instancia resoluble en un algoritmo polinomial se incrementa por un factor constante ($\sqrt{1000} \approxeq 31.62$ en el primer caso, $\sqrt[10]{1000} \approxeq 1.995$ en el segundo)

¿Qué representa en términos prácticos?

Suppose that computing power (in terms of computations per second) increases by a factor of 1000 at decade, and we have to choose between four different algorithms to find solutions for a critical application, which demands for an answer within a day. All of them can find solutions within a day for instances of size 100. Besides, it is known that complexity orders for these algorithms are: n^2, n^{10}, 10^n and $n!$ respectively.

year	pod. comp.	$O(n^2)$	$O(n^{10})$	$O(10^n)$	$O(n!)$
2020	1.000	100	100	100	100
2030	1.000.000	3,162	200	103	102
2040	10^9	100,000	398	106	104
2050	10^{12}	3,162,278	794	109	105
2060	10^{15}	100,000,000	1,585	112	106

Cuando el orden es exponencial, el tamaño de las instancias resolubles se vuelve "inmutable" a la potencia de cómputo a partir de instancias de cierto tamaño.

Problemas computacionalmente tratables

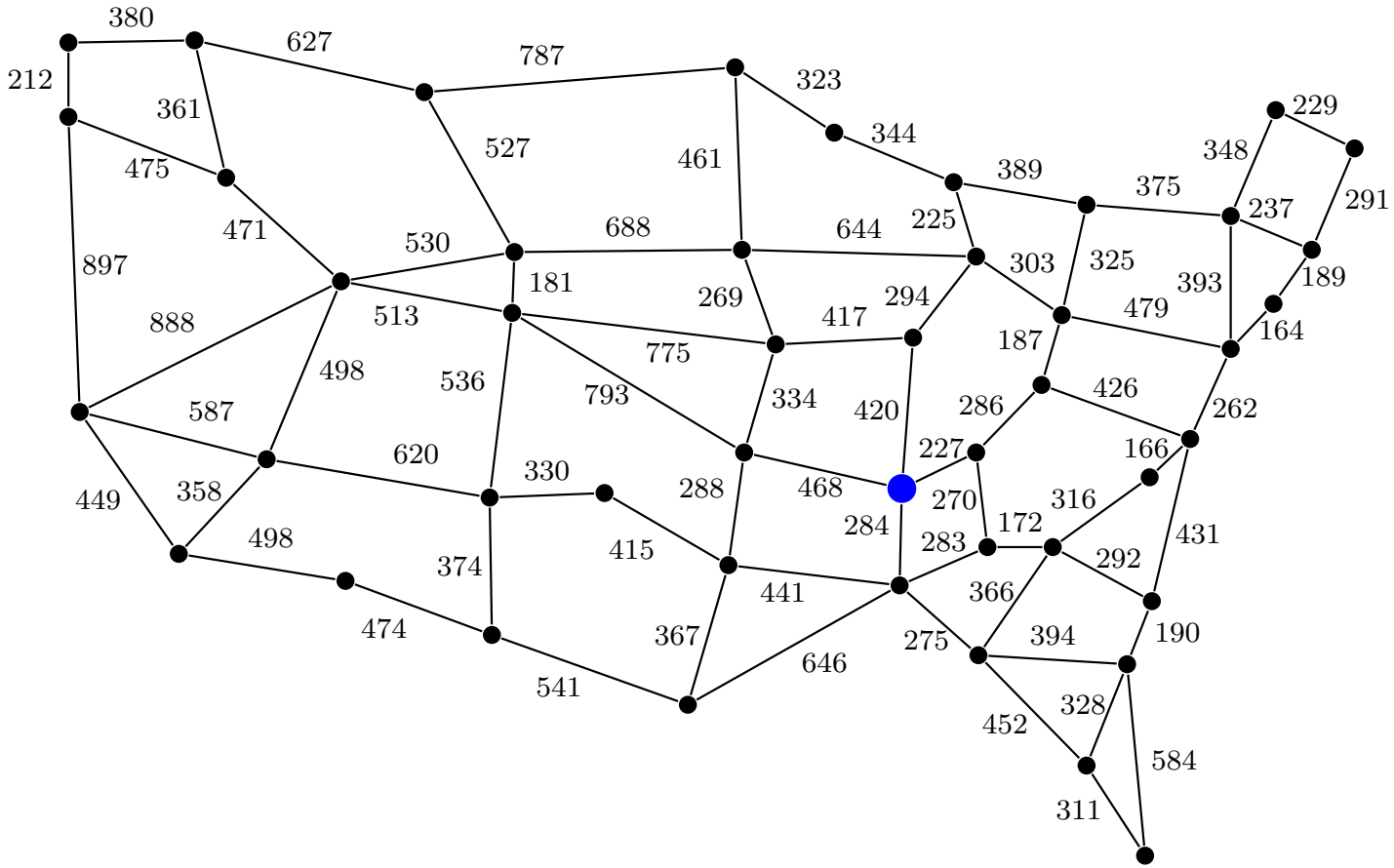
Definition

An algorithm is referred to as *efficient* or *tractable*, if and only if, its *asymptotic time complexity* is a monomial.
Formally, there is a $p \in \mathbb{N}$ such that the time required for the algorithm to find a solution ($f(x)$) is of *lower complexity order* than n^p (i.e. $f(x) = O(n^p)$).

In other words, for any input instance x of size n, the algorithm can find a solution in *polynomial time*.
Cualquier algoritmo polinomial para un problema ya es considerado una opción razonable. Aunque encontrado uno, se investiga intensamente cómo mejorarlo (bajarle el orden o la constante multiplicativa del término de mayor orden).
Un problema actual es que hay muchos "problemas importantes", para los cuales no se ha encontrado un algoritmo polinomial.

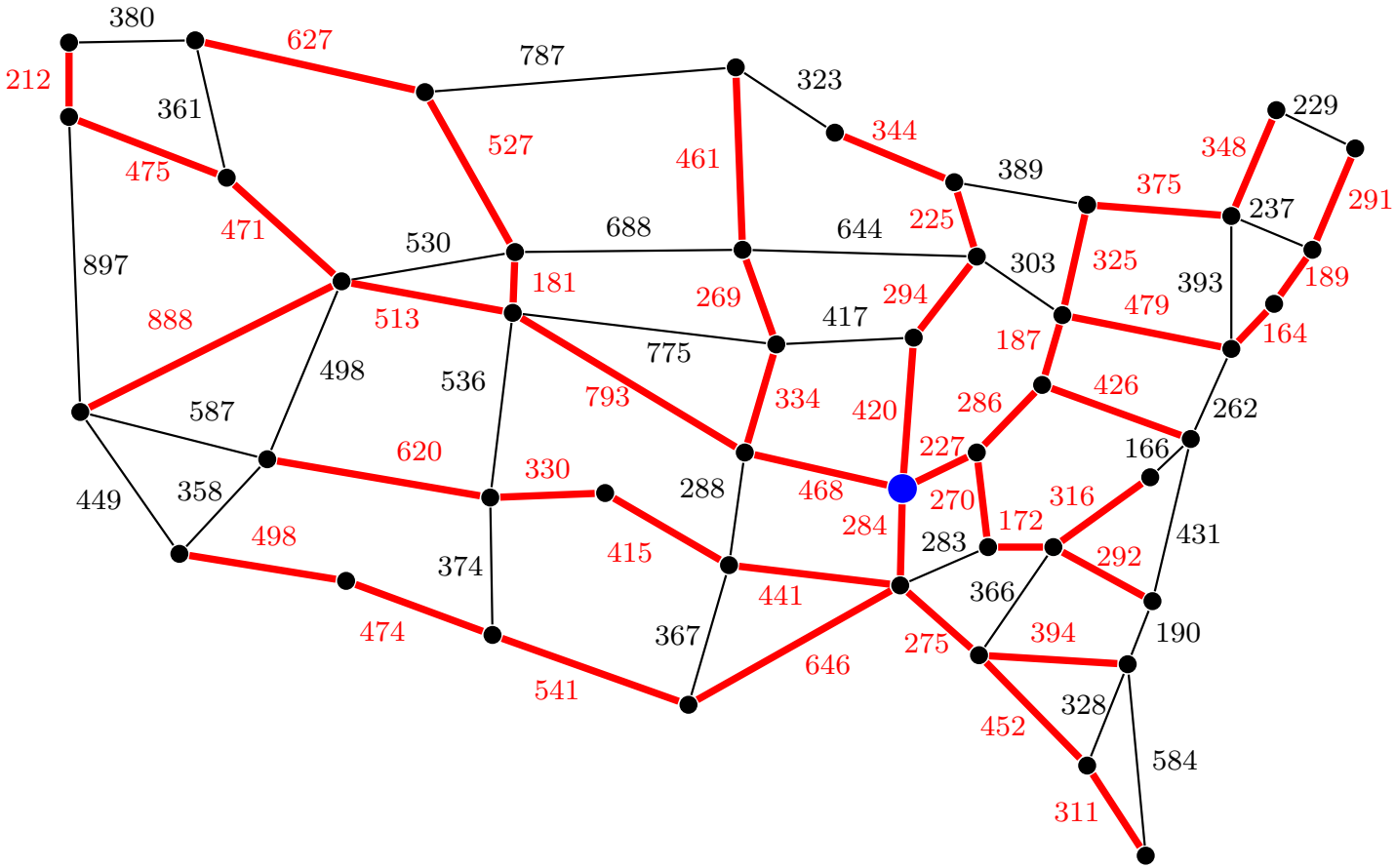
Problemas tratables/fáciles: Camino más corto (SPP)

¿Cuáles son los caminos más cortos desde el nodo azul a los demás, dentro USNet?



Problemas tratables/fáciles: Camino más corto (SPP)

¿Cuáles son los caminos más cortos desde el nodo azul a los demás, dentro USNet?



La solución es el árbol remarcado en la figura.

Problemas tratables/fáciles: Camino más corto (SPP)

El *Shortest Path Problem* tiene múltiples aplicaciones en:

- Problemas de ruteo (telecomunicaciones, transporte/GPS)
- Problemas de ordenamiento (planificación de actividades)
- Análisis de Redes Sociales
- Mantenimiento y remplazo de componentes

Se conocen muchos algoritmos eficientes según el tipo de grafo:

- Grafos no-dirigidos
- Grafos con aristas de igual peso
- Grafos dirigidos acíclicos
- Grafos dirigidos con pesos no-negativos
- Grafos planares
- Grafos dirigidos sin ciclos negativos

Problemas tratables/fáciles: Camino más corto (SPP)

Ejemplos de algoritmos más eficientes conocidos:

UNDIRECTED GRAPHS

- Dijkstra (1959), $O(V^2)$ cuando pesos en $\mathbb{R}^+$
- Fredman-Tarjan (1984) $O(E + V \log(V))$ con Fibonnacci Heap cuando pesos en $\mathbb{R}^+$
- Thorup (1999) $O(E)$ cuando pesos en $\mathbb{N}$

UNWEIGHTED GRAPHS

- Moore (1958) Breadth-first search (BFS), $O(E + V)$

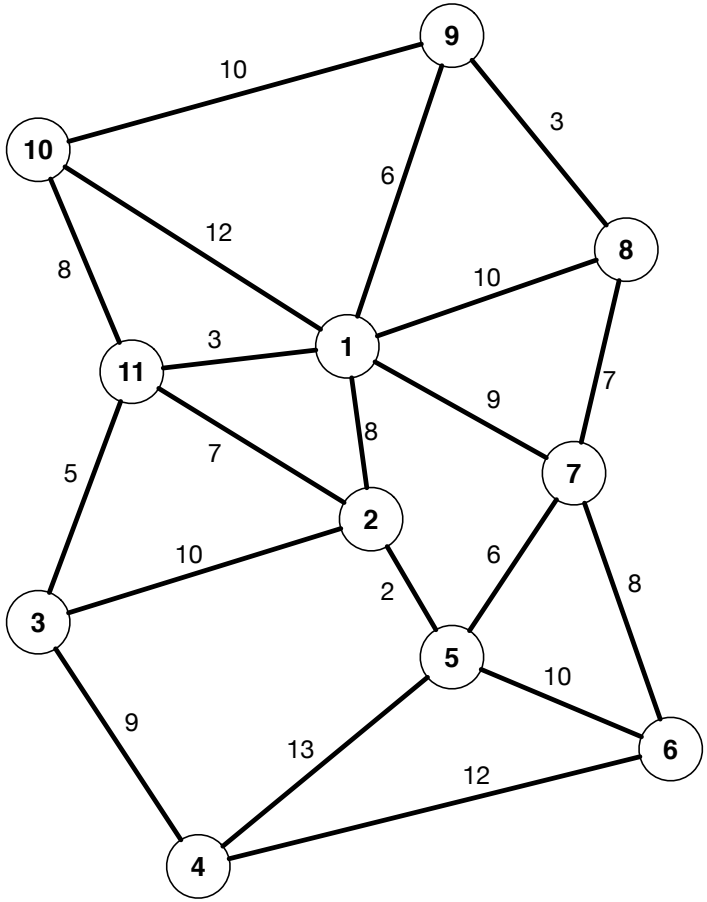
DG with arbitrary weights, without negative cycles

- Bellman-Ford (1958) / Moore (59), $O(VE)$

La lista completa es mucho más extensa.

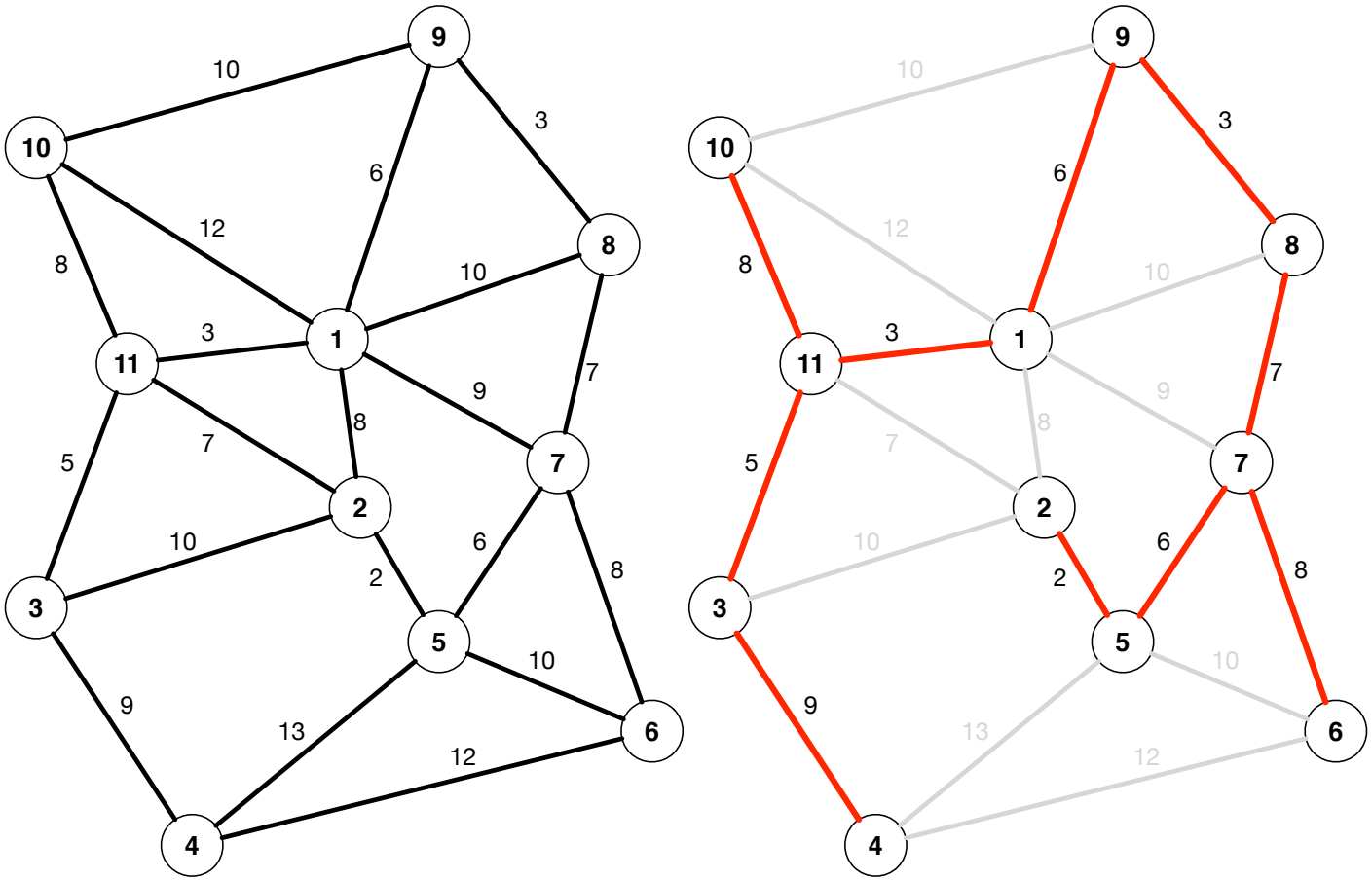
Problemas tratables/fáciles: Árbol de Cubrimiento (MST)

¿Cuál es un subgrafo conexo minimal (el árbol de cubrimiento) de costo mínimo para un grafo dado?



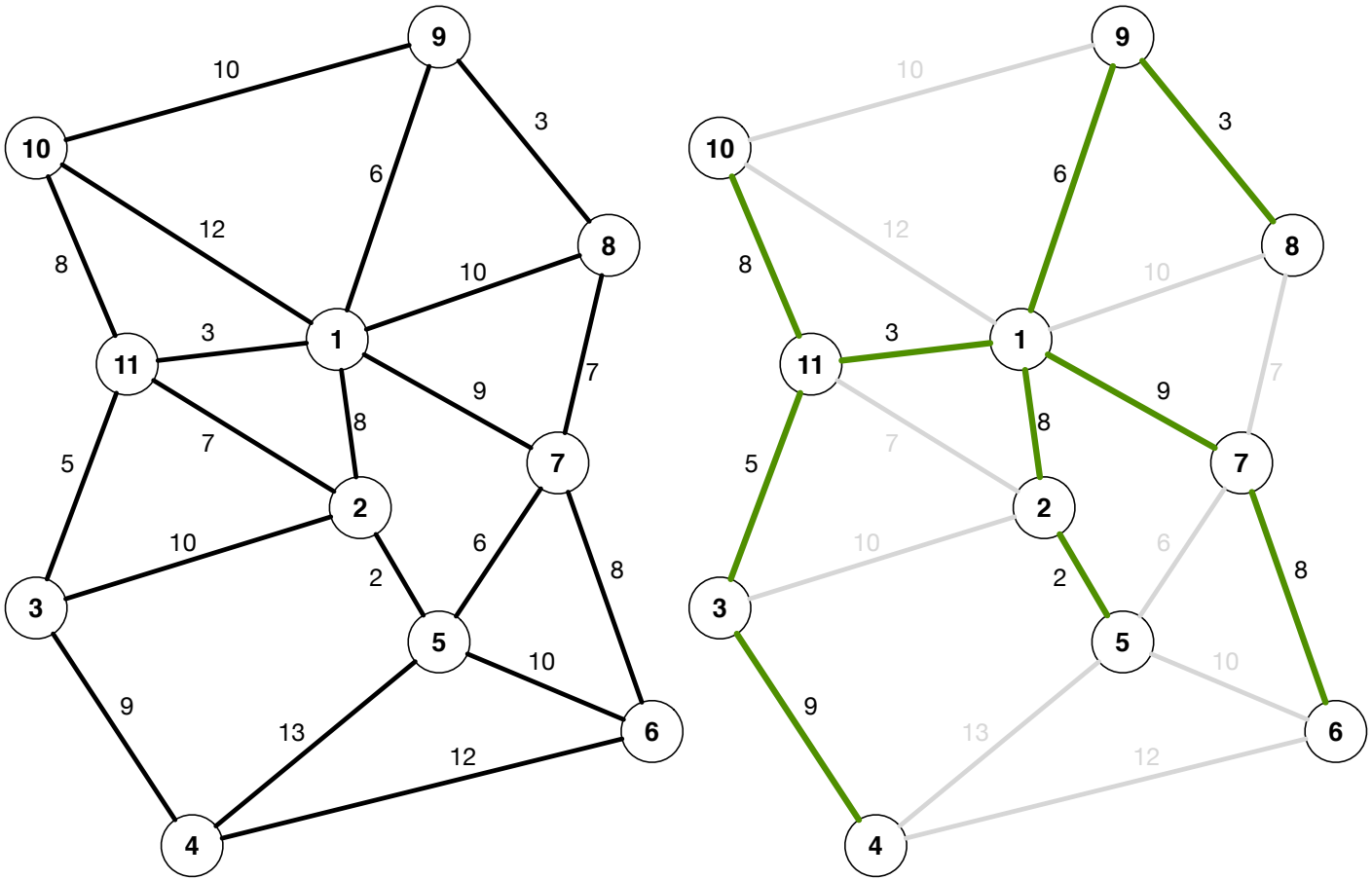
Problemas tratables/fáciles: Árbol de Cubrimiento (MST)

¿Cuál es un subgrafo conexo minimal (el árbol de cubrimiento) de costo mínimo para un grafo dado?



Problemas tratables/fáciles: Árbol de Cubrimiento (MST)

Observar que la solución (costo total 57) es distinta de la solución del SPP (costo 61).



Problemas tratables/fáciles: Árbol de Cubrimiento (MST)

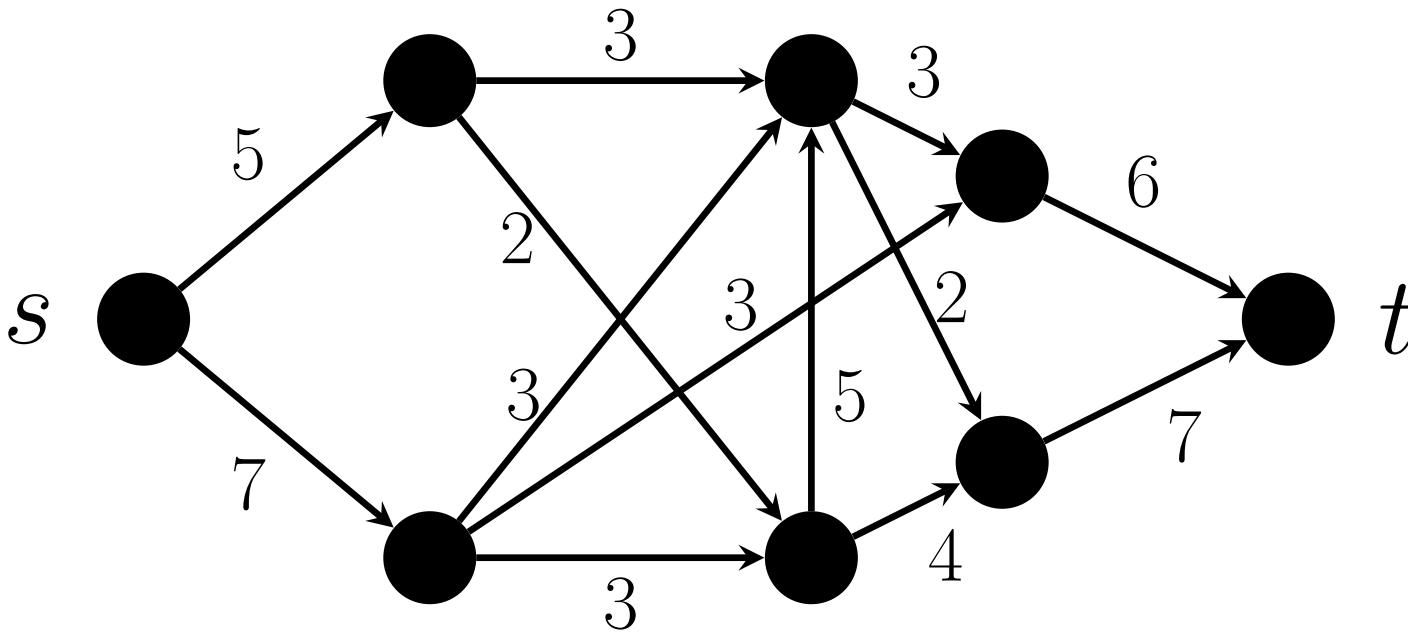
Un sub-grado de cubrimiento $T \subseteq G = (V, E)$ (no-dirigido), debe cumplir alguna de las siguientes condiciones para ser un árbol:

- T es conexo y acíclico
- T es conexo y minimal
- T es conexo y tiene $|V| - 1$ arcos
- T es acíclico y tiene $|V| - 1$ arcos
- Cualquier par de nodos u y v en T están conectados por un único camino

Los algoritmos polinomiales más famosos para resolver este problema son los de Prim (construye grafo conexo con $|V| - 1$ arcos) y Kruskal (acíclico con $|V| - 1$ arcos), cuyos órdenes de ejecución son $O(|E| + |V| \log |V|)$ y $O(|E| log |E|)$ respectivamente.

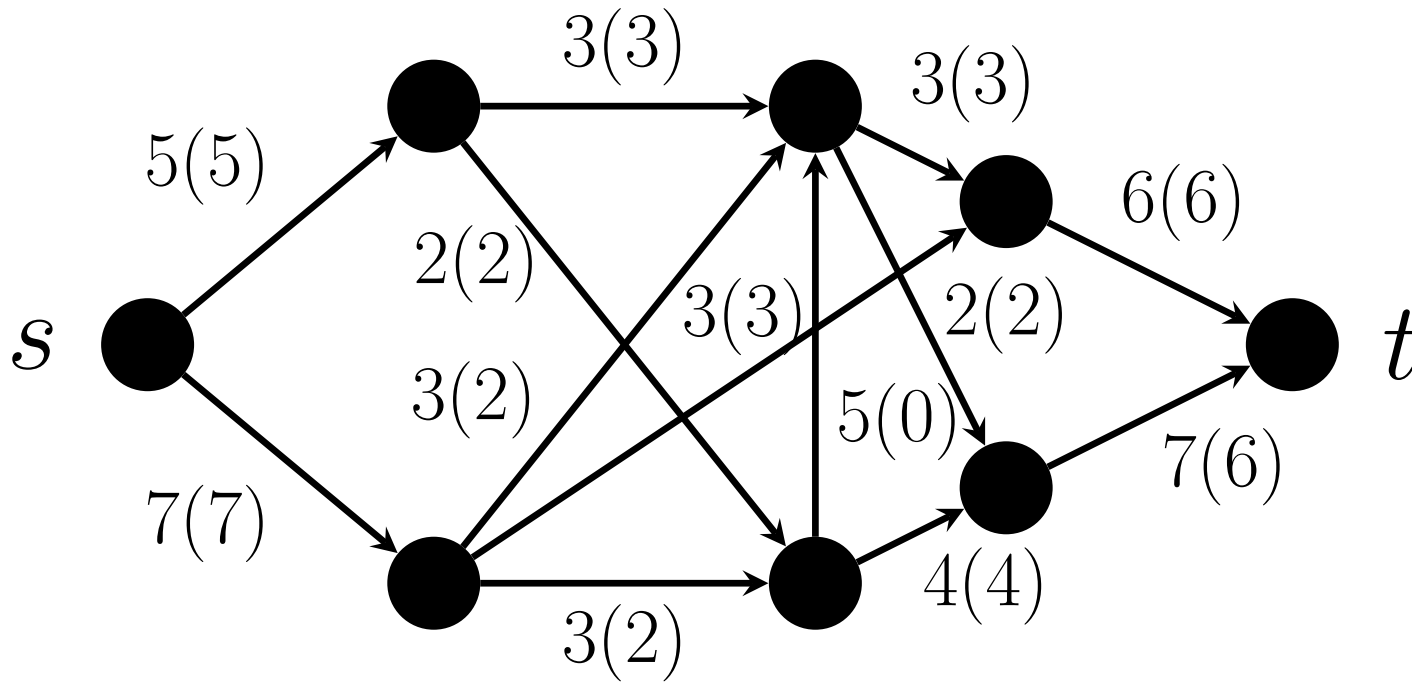
Problemas tratables/fáciles: Flujo Máximo (MFP)

¿Cuál es el máximo flujo que puede circular de s a t en esta red con esas capacidades?



Problemas tratables/fáciles: Flujo Máximo (MFP)

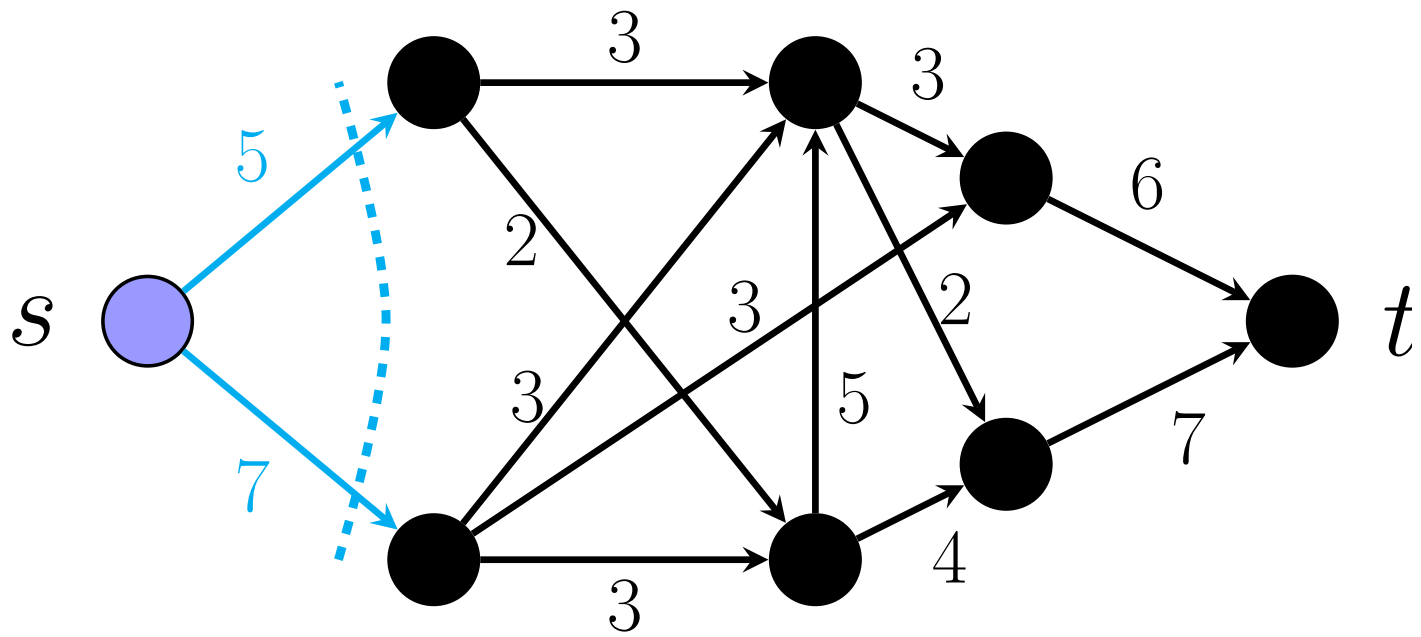
¿Cuál es el máximo flujo que puede circular de s a t en esta red con esas capacidades?



La respuesta es 12, y podría lograrse con esta configuración

Problemas tratables/fáciles: Flujo Máximo (MFP)

¿Cuál es el máximo flujo que puede circular de s a t en esta red con esas capacidades?

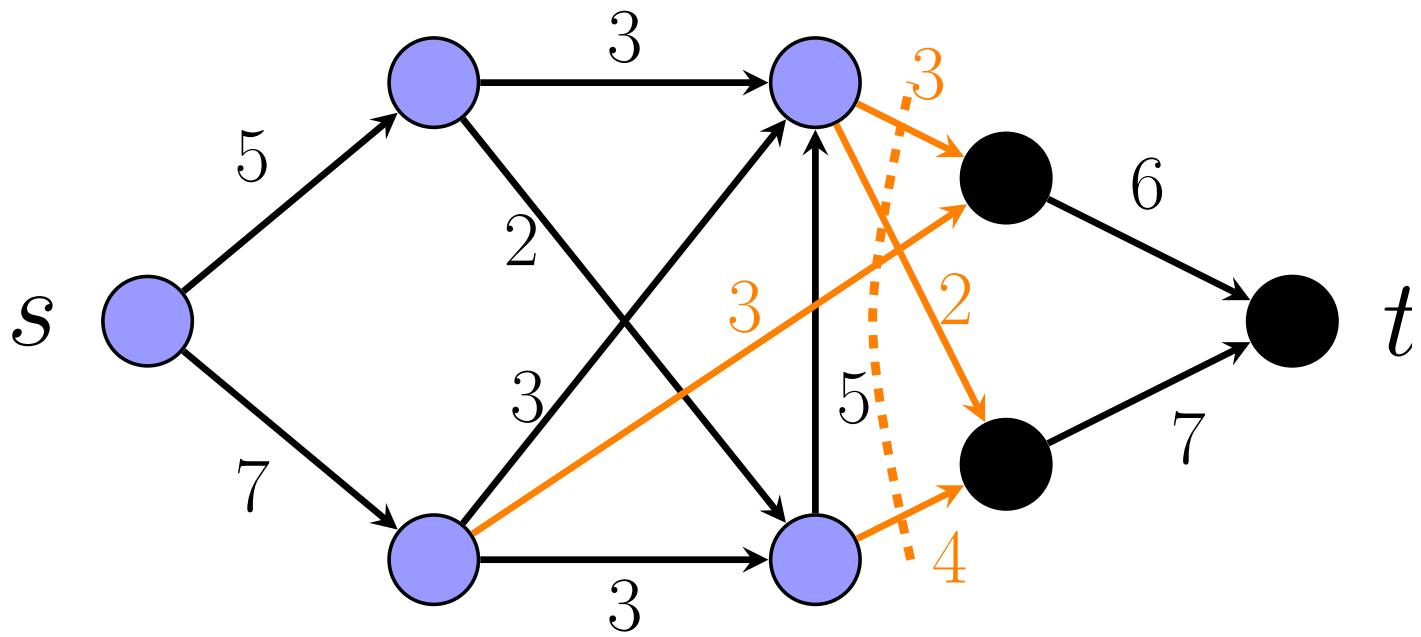


Theorem (Ford-Fulkerson (1962))

The maximum value of an s-t-flow is equal to the minimum capacity over all s-t-cuts.

Problemas tratables/fáciles: Flujo Máximo (MFP)

¿Cuál es el máximo flujo que puede circular de s a t en esta red con esas capacidades?



Theorem (Ford-Fulkerson (1962))

The maximum value of an s-t-flow is equal to the minimum capacity over all s-t-cuts.

Problemas tratables/fáciles: Flujo Máximo (MFP)

El *Maximum Flow Problem* tiene múltiples aplicaciones en:

- Logística y transporte (desde movimientos de mercancías a scheduling de aerolíneas)
- Resiliency (caminos disjuntos entre nodos, conjuntos de separación en una red)
- Cubrimientos jerárquicos (aplicaciones varias)

También hay muchas variantes y algoritmos:

Ford-Fulkerson (1956): Complejidad $O(E \max(f))$

Dinic (1970): Complejidad $O(VE \log(V))$

Malhotra, Pramodh, Kumar y Maheshwari (1978): $O(V^3)$

La lista completa es mucho más extensa.

Problemas tratables/fáciles: Flujo Máximo (MFP)

El *Maximum Flow Problem* tiene múltiples aplicaciones en:

- Logística y transporte (desde movimientos de mercancías a scheduling de aerolíneas)
- Resiliency (caminos disjuntos entre nodos, conjuntos de separación en una red)
- Cubrimientos jerárquicos (aplicaciones varias)

También hay muchas variantes y algoritmos:

Ford-Fulkerson (1956): Complejidad $O(E \max(f))$

Dinic (1970): Complejidad $O(VE \log(V))$

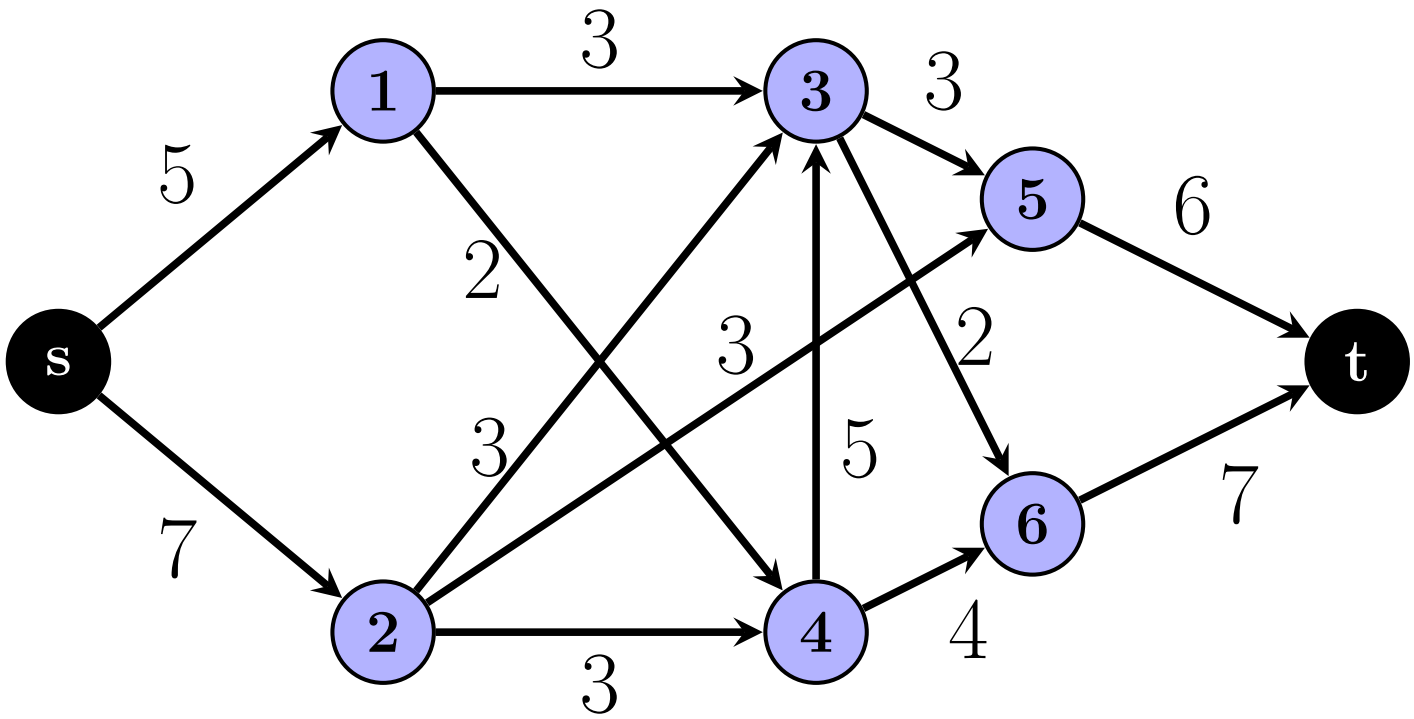
Malhotra, Pramodh, Kumar y Maheshwari (1978): $O(V^3)$

La lista completa es mucho más extensa.

¿Qué pasa con los problemas de optimización en general?

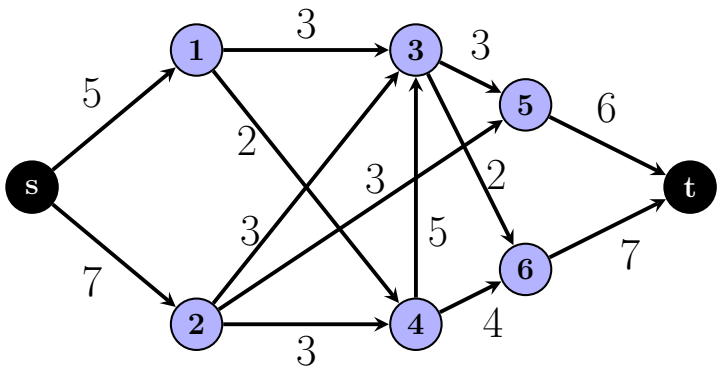
El Problema de Programación Lineal

Los problemas anteriormente citados admiten formulaciones LP (Linear Programming). Por ejemplo el MFP:



El Problema de Programación Lineal

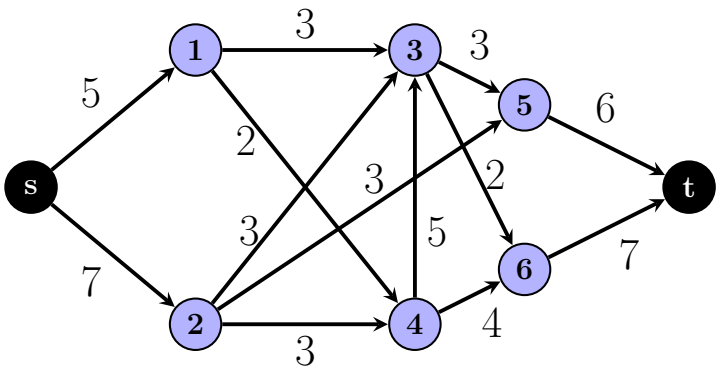
Los problemas anteriormente citados admiten formulaciones LP (Linear Programming). Por ejemplo el MFP:



$$
\begin{cases}
\max\ x_{s1} + x_{s2} \\
x_{s1} - x_{13} - x_{14} = 0 \\
x_{s2} - x_{23} - x_{24} - x_{25} = 0 \\
x_{13} + x_{23} + x_{43} - x_{35} - x_{36} = 0 \\
\dots \\
0 \leq x_{s1} \leq 5 \\
0 \leq x_{s2} \leq 7 \\
0 \leq x_{13} \leq 3 \\
\dots \\
0 \leq x_{6t} \leq 7
\end{cases}
$$

El Problema de Programación Lineal

Los problemas anteriormente citados admiten formulaciones LP (Linear Programming). Por ejemplo el MFP:

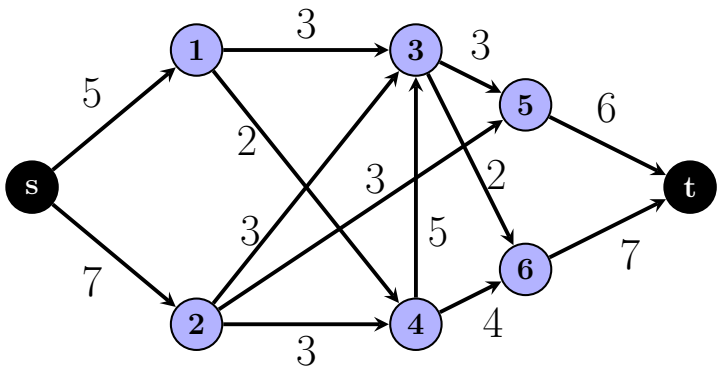


$$
\begin{cases}
\max\ x_{s1} + x_{s2} \\
x_{s1} - x_{13} - x_{14} = 0 \\
x_{s2} - x_{23} - x_{24} - x_{25} = 0 \\
x_{13} + x_{23} + x_{43} - x_{35} - x_{36} = 0 \\
\dots \\
0 \leq x_{s1} \leq 5 \\
0 \leq x_{s2} \leq 7 \\
0 \leq x_{13} \leq 3 \\
\dots \\
0 \leq x_{6t} \leq 7
\end{cases}
$$

Buscamos maximar el flujo que abandona s (y por tanto circula por el grafo)

El Problema de Programación Lineal

Los problemas anteriormente citados admiten formulaciones LP (Linear Programming). Por ejemplo el MFP:

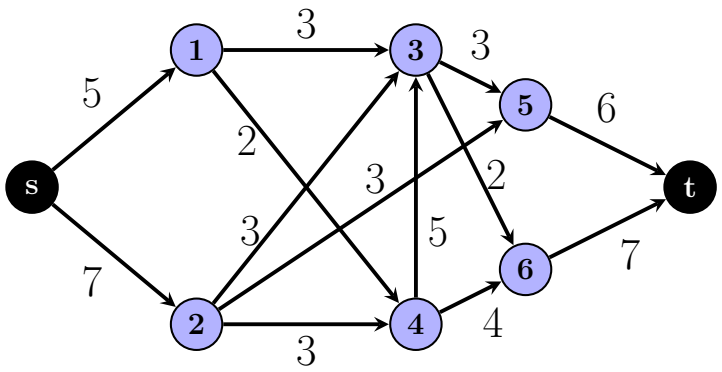


$$
\begin{cases}
\max\ x_{s1} + x_{s2} \\
x_{s1} - x_{13} - x_{14} = 0 \\
x_{s2} - x_{23} - x_{24} - x_{25} = 0 \\
x_{13} + x_{23} + x_{43} - x_{35} - x_{36} = 0 \\
\dots \\
0 \leq x_{s1} \leq 5 \\
0 \leq x_{s2} \leq 7 \\
0 \leq x_{13} \leq 3 \\
\dots \\
0 \leq x_{6t} \leq 7
\end{cases}
$$

Asegurando balance de flujo en los nodos intermedios (todos menos s y t)

El Problema de Programación Lineal

Los problemas anteriormente citados admiten formulaciones LP (Linear Programming). Por ejemplo el MFP:

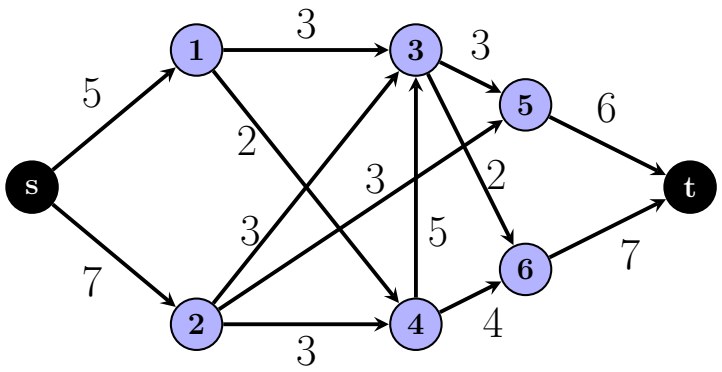


$$
\begin{cases}
\max\ x_{s1} + x_{s2} \\
x_{s1} - x_{13} - x_{14} = 0 \\
x_{s2} - x_{23} - x_{24} - x_{25} = 0 \\
x_{13} + x_{23} + x_{43} - x_{35} - x_{36} = 0 \\
\dots \\
0 \leq x_{s1} \leq 5 \\
0 \leq x_{s2} \leq 7 \\
0 \leq x_{13} \leq 3 \\
\dots \\
0 \leq x_{6t} \leq 7
\end{cases}
$$

Imponiendo al mismo tiempo que el valor del flujo en cada arco (x_{ij}) no viole la capacidad del enlace

El Problema de Programación Lineal

Los problemas anteriormente citados admiten formulaciones LP (Linear Programming). Por ejemplo el MFP:



$$
\begin{cases}
\max\ x_{s1} + x_{s2} \\
x_{s1} - x_{13} - x_{14} = 0 \\
x_{s2} - x_{23} - x_{24} - x_{25} = 0 \\
x_{13} + x_{23} + x_{43} - x_{35} - x_{36} = 0 \\
\dots \\
0 \leq x_{s1} \leq 5 \\
0 \leq x_{s2} \leq 7 \\
0 \leq x_{13} \leq 3 \\
\dots \\
0 \leq x_{6t} \leq 7
\end{cases}
$$

¡¡También puede verse que el dual de este problema, corresponde a encontrar el corte de capacidad mínima!!

El Problema de Programación Lineal

El SPP se puede escribir como una variante del problema anterior: imponiendo capacidad 1 en todos los arcos, un flujo de valor 1 entre los nodos origen/destino, y minimizando $\sum_{ij \in E} c_{ij} x_{ij}$.
Para el MST se puede formular así:

$$\begin{cases} \min \sum_{ij \in E} c_{ij} x_{ij} \\ \sum_{ij \in E} x_{ij} = |V| - 1 \\ \sum_{\substack{ij \in E \\ i,j \in S}} x_{ij} \leq |S| - 1, \quad \forall S \subseteq V, \\ x_{ij} \geq 0 \end{cases}$$

Construye un subgrafo acíclico de $|V| - 1$ arcos. Para ser auto-contenido se necesita un número exponencial de restricciones.

El Problema de Programación Lineal

El SPP se puede escribir como una variante del problema anterior: imponiendo capacidad 1 en todos los arcos, un flujo de valor 1 entre los nodos origen/destino, y minimizando $\sum_{ij \in E} c_{ij} x_{ij}$.
También se puede formular así:

$$\begin{cases} \min \sum_{ij \in E} c_{ij} x_{ij} \\ \sum_{ij \in E} x_{ij} = |V| - 1 \\ \sum_{\substack{ij \in E \\ ij \in \delta(S)}} x_{ij} \geq 1, \qquad \forall S \subseteq V, \emptyset \neq S \neq V \\ x_{ij} \in \{0, 1\} \end{cases}$$

Donde $\delta(S)$ son los arcos de S a S^c. Se construye un subgrafo conexo de $|V| - 1$ arcos.

El Problema de Programación Lineal

El SPP se puede escribir como una variante del problema anterior: imponiendo capacidad 1 en todos los arcos, un flujo de valor 1 entre los nodos origen/destino, y minimizando $\sum_{ij \in E} c_{ij} x_{ij}$.
También se puede formular así:

$$\begin{cases} \min \sum_{ij \in E} c_{ij} x_{ij} \\ \sum_{ij \in E} x_{ij} = |V| - 1 \\ \sum_{\substack{ij \in E \\ ij \in \delta(S)}} x_{ij} \geq 1, \qquad \forall S \subseteq V, \emptyset \neq S \neq V \\ x_{ij} \in \{0, 1\} \end{cases}$$

Tiene el problema adicional de que la relajación entera no funciona en este caso.

El Problema de Programación Lineal

El SPP se puede escribir como una variante del problema anterior: imponiendo capacidad 1 en todos los arcos, un flujo de valor 1 entre los nodos origen/destino, y minimizando $\sum_{ij \in E} c_{ij} x_{ij}$.

Ejercicio: Proponer otra formulación para el MST basada en un problema de flujo en redes.

Complejidad intrínseca en problemas de decisión

Definition

Given a decision problem π, we call an *instance* of it to a concrete set of parameters, which univocally feed an algorithm in order to find an answer (Yes or No). Any decision problem π has an associated domain-set of instances D_π, where the problem makes sense. Let $Y_\pi \subseteq D_\pi$ be those instances for which the answer is *Yes*.

Definition

We say a decision problem π is of class **P**, if and only if, there is an algorithm capable of finding answers (solutions) in polynomial time as a function of the instance size.
We say a decision problem π is of class **NP**, if and only if, there is an algorithm capable of checking solutions (given by an external oracle) in polynomial time as a function of the instance size.

Complejidad intrínseca en problemas de decisión

¿Pertenecen a P o NP los problemas SPP, MST o MFP?

Complejidad intrínseca en problemas de decisión

¿Pertenecen a P o NP los problemas SPP, MST o MFP?
Sí, los tres están en **P** porque existen algoritmos de orden polinomial en el tamaño del problema (V), que permiten encontrar la solución (e.g. Dijkstra, Prim, Ford-Fulkerson)

Complejidad intrínseca en problemas de decisión

¿Pertenecen a P o NP los problemas SPP, MST o MFP?
Sí, los tres están en **P** porque existen algoritmos de orden polinomial en el tamaño del problema (V), que permiten encontrar la solución (e.g. Dijkstra, Prim, Ford-Fulkerson)
Sabiendo que SPP y MFP (también MST) pueden formularse como problemas LP ¿Podemos decir que LP está en P?

Complejidad intrínseca en problemas de decisión

¿Pertenecen a P o NP los problemas SPP, MST o MFP?
Sí, los tres están en **P** porque existen algoritmos de orden polinomial en el tamaño del problema (V), que permiten encontrar la solución (e.g. Dijkstra, Prim, Ford-Fulkerson)
Sabiendo que SPP y MFP (también MST) pueden formularse como problemas LP ¿Podemos decir que LP está en P?
No, eso sólo nos dice que *algunas instancias* pueden resolverse en tiempo polinomial. Para estar en **P** hay que tener certeza que cualquier instancia puede resolverse en tiempo polinomial

Complejidad intrínseca en problemas de decisión

¿Pertenecen a P o NP los problemas SPP, MST o MFP?
Sí, los tres están en **P** porque existen algoritmos de orden polinomial en el tamaño del problema (V), que permiten encontrar la solución (e.g. Dijkstra, Prim, Ford-Fulkerson)
Sabiendo que SPP y MFP (también MST) pueden formularse como problemas LP ¿Podemos decir que LP está en P?
No, eso sólo nos dice que *algunas instancias* pueden resolverse en tiempo polinomial. Para estar en **P** hay que tener certeza que cualquier instancia puede resolverse en tiempo polinomial
¿Se cumple que P$\subseteq$NP?

Complejidad intrínseca en problemas de decisión

¿Pertenecen a P o NP los problemas SPP, MST o MFP?
Sí, los tres están en **P** porque existen algoritmos de orden polinomial en el tamaño del problema (V), que permiten encontrar la solución (e.g. Dijkstra, Prim, Ford-Fulkerson)
Sabiendo que SPP y MFP (también MST) pueden formularse como problemas LP ¿Podemos decir que LP está en P?
No, eso sólo nos dice que *algunas instancias* pueden resolverse en tiempo polinomial. Para estar en **P** hay que tener certeza que cualquier instancia puede resolverse en tiempo polinomial
¿Se cumple que P$\subseteq$NP?
Sí, en el peor de los casos se resuelve el problema (es polinomial) y se comparan los resultados

Complejidad intrínseca en problemas de decisión

¿Pertenecen a P o NP los problemas SPP, MST o MFP?
Sí, los tres están en **P** porque existen algoritmos de orden polinomial en el tamaño del problema (V), que permiten encontrar la solución (e.g. Dijkstra, Prim, Ford-Fulkerson)
Sabiendo que SPP y MFP (también MST) pueden formularse como problemas LP ¿Podemos decir que LP está en P?
No, eso sólo nos dice que *algunas instancias* pueden resolverse en tiempo polinomial. Para estar en **P** hay que tener certeza que cualquier instancia puede resolverse en tiempo polinomial
¿Se cumple que P$\subseteq$NP?
Sí, en el peor de los casos se resuelve el problema (es polinomial) y se comparan los resultados
¿Se cumple que NP$\subseteq$P?

Complejidad intrínseca en problemas de decisión

¿Pertenecen a P o NP los problemas SPP, MST o MFP?
Sí, los tres están en **P** porque existen algoritmos de orden polinomial en el tamaño del problema (V), que permiten encontrar la solución (e.g. Dijkstra, Prim, Ford-Fulkerson)
Sabiendo que SPP y MFP (también MST) pueden formularse como problemas LP ¿Podemos decir que LP está en P?
No, eso sólo nos dice que *algunas instancias* pueden resolverse en tiempo polinomial. Para estar en **P** hay que tener certeza que cualquier instancia puede resolverse en tiempo polinomial
¿Se cumple que P$\subseteq$NP?
Sí, en el peor de los casos se resuelve el problema (es polinomial) y se comparan los resultados
¿Se cumple que NP$\subseteq$P?
No se sabe. Éste es de hecho uno de los problemas abiertos más importantes de la informática

Complejidad intrínseca en problemas de decisión

The **Number Partition Problem (NPP)** consists in finding a two subsets partition with almost the same sum for a known multiset of numbers $\mathcal{M} = \{a_1, \ldots, a_N\}$. Buscamos $\mathcal{A} \subset \{1, \ldots, N\}$ tal que se cumpla: $|\sum_{i \in \mathcal{A}} a_i - \sum_{i \notin \mathcal{A}} a_i| \leq 1$ ¿Está en **NP** el NPP?

Complejidad intrínseca en problemas de decisión

The **Number Partition Problem (NPP)** consists in finding a two subsets partition with almost the same sum for a known multiset of numbers $\mathcal{M} = \{a_1, \ldots, a_N\}$. Buscamos $\mathcal{A} \subset \{1, \ldots, N\}$ tal que se cumpla: $|\sum_{i \in \mathcal{A}} a_i - \sum_{i \notin \mathcal{A}} a_i| \leq 1$ ¿Está en **NP** el NPP?
No es fácil partir $\{62, 83, 121, 281, 486, 734, 771, 854, 885, 1003\}$

Complejidad intrínseca en problemas de decisión

The **Number Partition Problem (NPP)** consists in finding a two subsets partition with almost the same sum for a known multiset of numbers $\mathcal{M} = \{a_1, \ldots, a_N\}$. Buscamos $\mathcal{A} \subset \{1, \ldots, N\}$ tal que se cumpla: $|\sum_{i \in \mathcal{A}} a_i - \sum_{i \notin \mathcal{A}} a_i| \leq 1$ ¿Está en **NP** el NPP?
No es fácil partir $\{62, 83, 121, 281, 486, 734, 771, 854, 885, 1003\}$ pero sí lo es verificar que $\{281, 734, 771, 854\}$, $\{62, 83, 121, 486, 885, 1003\}$ es una partición que satisface el NPP.

Complejidad intrínseca en problemas de decisión

The **Number Partition Problem (NPP)** consists in finding a two subsets partition with almost the same sum for a known multiset of numbers $\mathcal{M} = \{a_1, \ldots, a_N\}$. Buscamos $\mathcal{A} \subset \{1, \ldots, N\}$ tal que se cumpla: $|\sum_{i \in \mathcal{A}} a_i - \sum_{i \notin \mathcal{A}} a_i| \leq 1$ ¿Está en **NP** el NPP?
No es fácil partir $\{62, 83, 121, 281, 486, 734, 771, 854, 885, 1003\}$ pero sí lo es verificar que $\{281, 734, 771, 854\}$, $\{62, 83, 121, 486, 885, 1003\}$ es una partición que satisface el NPP.

The **Boolean Satisfiability Problem (SAT)** is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. Por ejemplo, encontrar valores $x_i \in \{0, 1\}$, $i = 1, 2, 3$, que verifquen $(x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge x_3 = 1$.
¿Es **NP** el SAT?

Complejidad intrínseca en problemas de decisión

The **Number Partition Problem (NPP)** consists in finding a two subsets partition with almost the same sum for a known multiset of numbers $\mathcal{M} = \{a_1, \ldots, a_N\}$. Buscamos $\mathcal{A} \subset \{1, \ldots, N\}$ tal que se cumpla: $|\sum_{i \in \mathcal{A}} a_i - \sum_{i \notin \mathcal{A}} a_i| \leq 1$ ¿Está en **NP** el NPP?
No es fácil partir $\{62, 83, 121, 281, 486, 734, 771, 854, 885, 1003\}$ pero sí lo es verificar que $\{281, 734, 771, 854\}$, $\{62, 83, 121, 486, 885, 1003\}$ es una partición que satisface el NPP.

The **Boolean Satisfiability Problem (SAT)** is the problem of determining if there exists an interpretation that satisfies a given Boolean formula. Por ejemplo, encontrar valores $x_i \in \{0, 1\}$, $i = 1, 2, 3$, que verifquen $(x_1 \vee \neg x_3) \wedge (\neg x_1 \vee x_2) \wedge x_3 = 1$.
¿Es **NP** el SAT? Sí. Es fácil ver que $x_1 = x_2 = x_3 = 1$ es solución.

Una regla para comparar las complejidades

Los padres de la Teoría de la Complejidad incluyen a: Alan Turing, Stephen Cook and Richard Karp. La teoría propone una forma de caracterizar/comparar problemas.

Definition

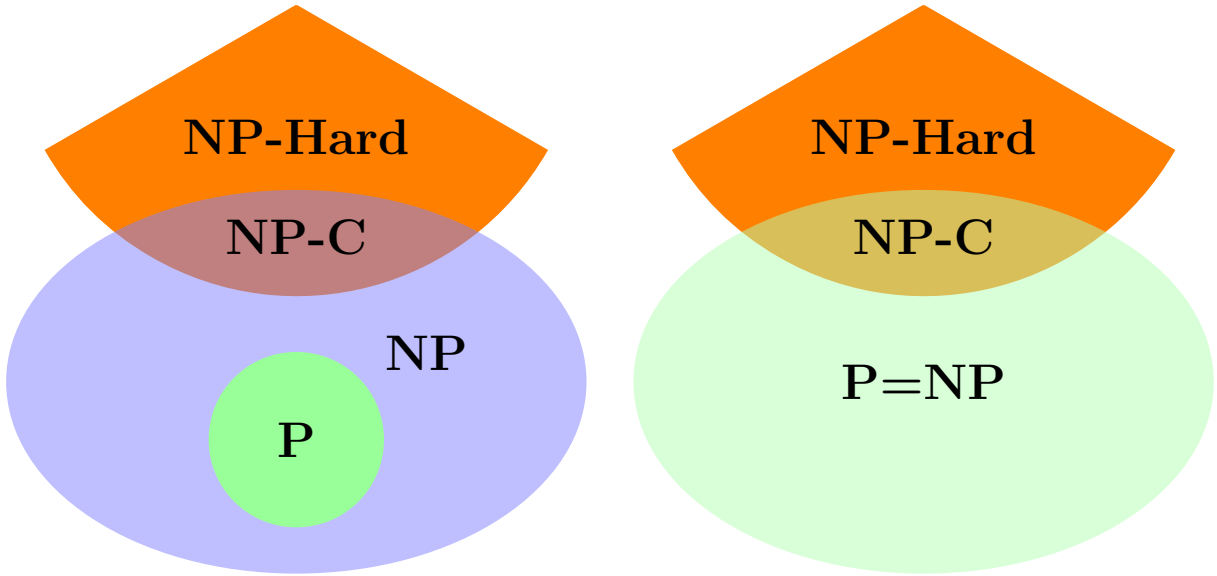
Given any two decision problems π and π', and being D_π and $D_{\pi'}$ their respective sets of instances, we call *polynomial reduction* of π' to π ($\pi' \preccurlyeq \pi$) to any function $f : D_{\pi'} \to D_\pi$ of polynomial complexity, such that for all $d \in D_{\pi'}$, it fulfills that $d \in Y_{\pi'}$ if and only if $f(d) \in Y_\pi$.

The existence of a polynomial reduction from π' to π ($\pi' \preccurlyeq \pi$) means that if anyone develops an efficient algorithm to find solutions to any instance of π, through the reduction process it can be used as the kernel to construct an efficient algorithm to find solutions to any instance of π'.

Clasificación de problemas según su complejidad conocida

Definition

Given a problem π we say that it is *NP-Hard* if and only if for all $\pi' \in NP$ it holds $\pi' \preccurlyeq \pi$. When besides this, it holds that $\pi \in NP$, we say π is *NP-Complete* or *NP-C*.



Cuál es el diagrama correcto depende de si $\mathbf{P} \neq \mathbf{NP}$ o $\mathbf{P} = \mathbf{NP}$.
Se conjetura que $\mathbf{P} \neq \mathbf{NP}$.

Clasificación de problemas según su complejidad conocida

Theorem (Cook's (1971))

The boolean SATisfiability (SAT) problem is NP-Complete, that is: the SAT is as hard as any other problem of NP.

On 1972 Richard Karp started the construction of a list of 21 polynomial reductions of SAT to other *NP* problems, thereby showing that all of them are *NP-complete* (Karp's list).

- Determining if there is a cut-set of certain size for a graph is *NP-Complete*, so it is finding a "Maximum Cut Set"
- The "Number Partitioning Problem" is *NP-Complete*.
- The Travelling Salesman Problem (TSP) is *NP-Hard* in general. That is: given a list of cities and distances between each pair of them, what is the cycle of lowest length that spans all cities. If distances are integer numbers, the TSP turns *NP-Complete*.

Clasificación de problemas según su complejidad conocida

The standard procedure to prove that a problem $\pi \in NP$ is *NP-Complete*, consists in finding a well known *NP-complete* problem (π') and a polynomial reductions from π' to π (i.e. proving that $\pi' \preccurlyeq \pi$).

Hence, the transitivity of "$\preccurlyeq$" guarantees that: $SAT \preccurlyeq \pi$.

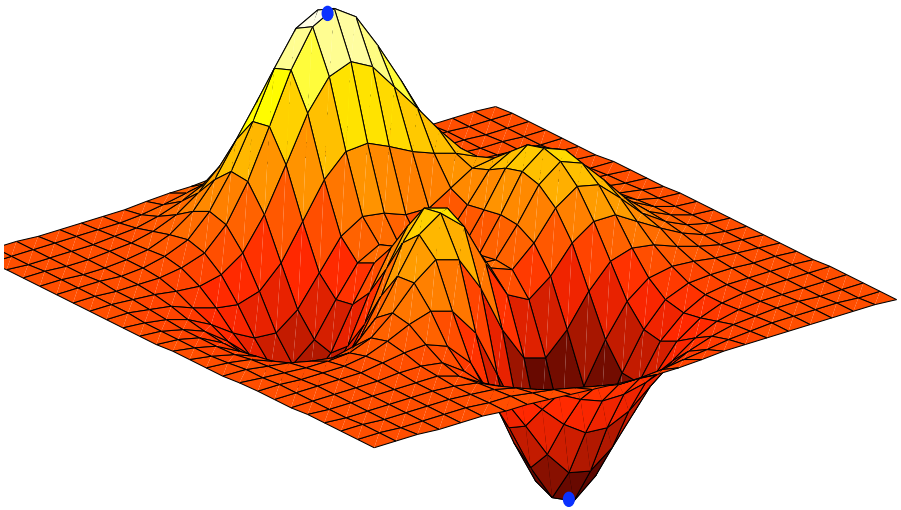
Complementarily and since SAT is the hardest NP problem (Cook's theorem), both complexities are equivalent and π is *NP-Complete*.

It is worth pointing out that the previous procedure guarantees π is *NP-Hard*, even if we cannot prove $\pi \in NP$.

El problema general de optimización

Definition

Given a domain X for a set of n variables (e.g. $X = \mathbb{R}^n$, $X = \mathbb{N}^n$ or $X = \{0,1\}^n$), an objective function $f : X \to \mathbb{R}$ and a set of m constraints to be fulfilled $g : X \to \mathbb{R}^m$, an optimization problem consists in finding $\bar{x} \in X$, such that $f(\bar{x})$ is the minimum (or maximum) value of f while $g(\bar{x}) \leq 0$.



$$(P) \begin{cases} \min f(x) \\ g(x) \leq 0 \\ x \in X \end{cases}$$

El problema general de optimización

El problema de optimización cuadrática sin restricciones está en **P**, porque el Método del Gradiente Conjugado (Magnus Hestenes y Eduard Stiefel, 1952) encuentra el óptimo en n pasos.

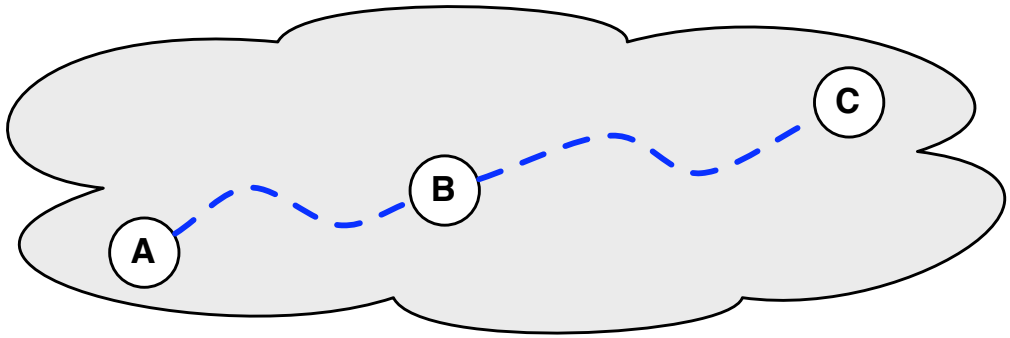
On 1984 Narendra Karmarkar proved that when $X = \mathbb{R}^n$, g is a linear function and f is also linear (or even quadratic), (P) can be solved in polynomial time. So Linear Programming (LP) problems are computationally easy to solve (i.e. they are in **P** class).

However, when $X \subseteq \mathbb{Z}^n$ the problem (P) is hard in general, even when f and g are linear. We call an optimization problem as a *combinatorial optimization problem* when all variables are of integer type.

This extends to cases where only some of the variables are of integer type whereas the others are real numbers. These problems are called Mixed Integer Programming (MIP) problems. In general MIP problems are hard to solve.

El Principio de Optimalidad

Muchos de los problemas tratables en forma exacta cumplen el *Principio de Optimalidad* (Bellman): "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision".



En el SPP, el Principio de Optimalidad representa que si se conoce el camino más corto entre dos nodos (A y C), y el nodo B pertenece al mismo, no hay un camino más corto para conectar a A y B (o B y C), que el subcamino correspondiente en el original.

Método de Programación Dinámica (Bellman 1940-1953)

- Es un *método recursivo* para resolver problemas complejos mediante su descomposición en problemas más simples
- Durante la recursión se almacenan los resultados encontrados al momento, y luego se usan en los problemas más complejos
- Deben existir "pasos base", instancias de tamaño reducido para las que la solución es trivial o conocida
- Se usa en problemas de: matemáticas, management, economía, computer science y bioinformática
- Requiere una estructura particular del problema, con un conjunto discreto de etapas y estados en cada etapa
- El problema debe cumplir el Principio de Optimalidad entre sus etapas, lo que permite un planteo *greedy*
- Debe ser Markoviano cuando es estocástico

Método de Programación Dinámica (ejemplo en juegos)

Suponga una mesa con 30 cerillos. En cada turno, un jugador puede tomar 1, 2 o 3 cerillos. Se continúa hasta que un jugador levante el último, en cuyo caso pierde ¿Cómo jugar para ganar?

Método de Programación Dinámica (ejemplo en juegos)

Suponga una mesa con 30 cerillos. En cada turno, un jugador puede tomar 1, 2 o 3 cerillos. Se continúa hasta que un jugador levante el último, en cuyo caso pierde ¿Cómo jugar para ganar?

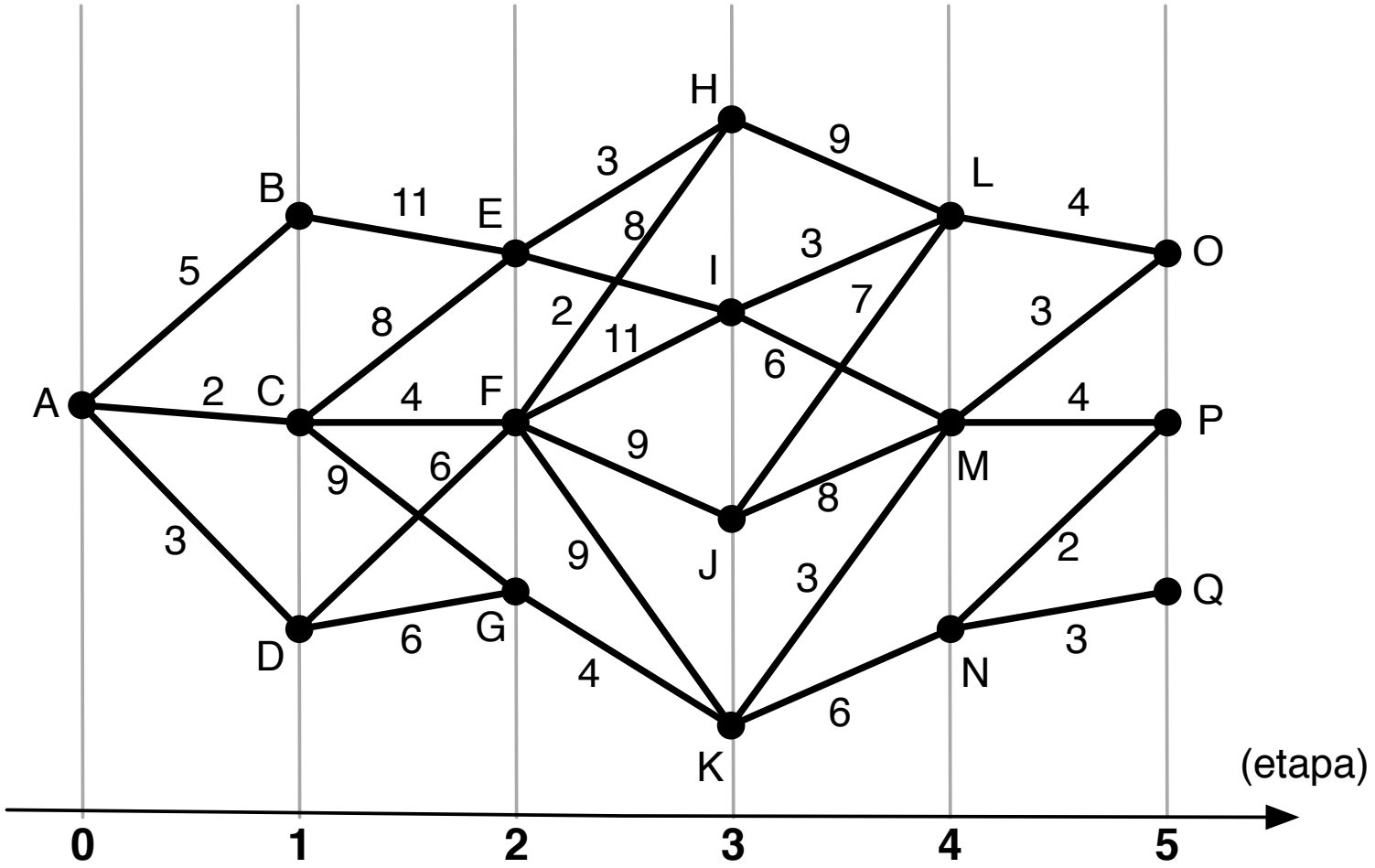
Hay que forzar al otro a llegar a una mano con un cerillo. Gano si llego a una mano con $\{2,3,4\}$ cerillos. La anterior entonces debe ser de 5 para el oponente. Puedo forzar esto (y ganar) si llego a una mano con $\{6,7,8\}$ cerillos. Su mano anterior entonces debe ser de 9. Puedo forzar esa mano si llego a una con $\{10,11,12\}$.

La secuencia (él,yo) sigue así: $(13,\{14,15,16\})$, $(17,\{18,19,20\})$, $(21,\{22,23,24\})$, $(25,\{26,27,28\})$. El conjunto prohibido es $\{1,5,9,13,17,21,25,29\}$.

¡¡Gana siempre el que saca un solo cerillo en la primer mano!!

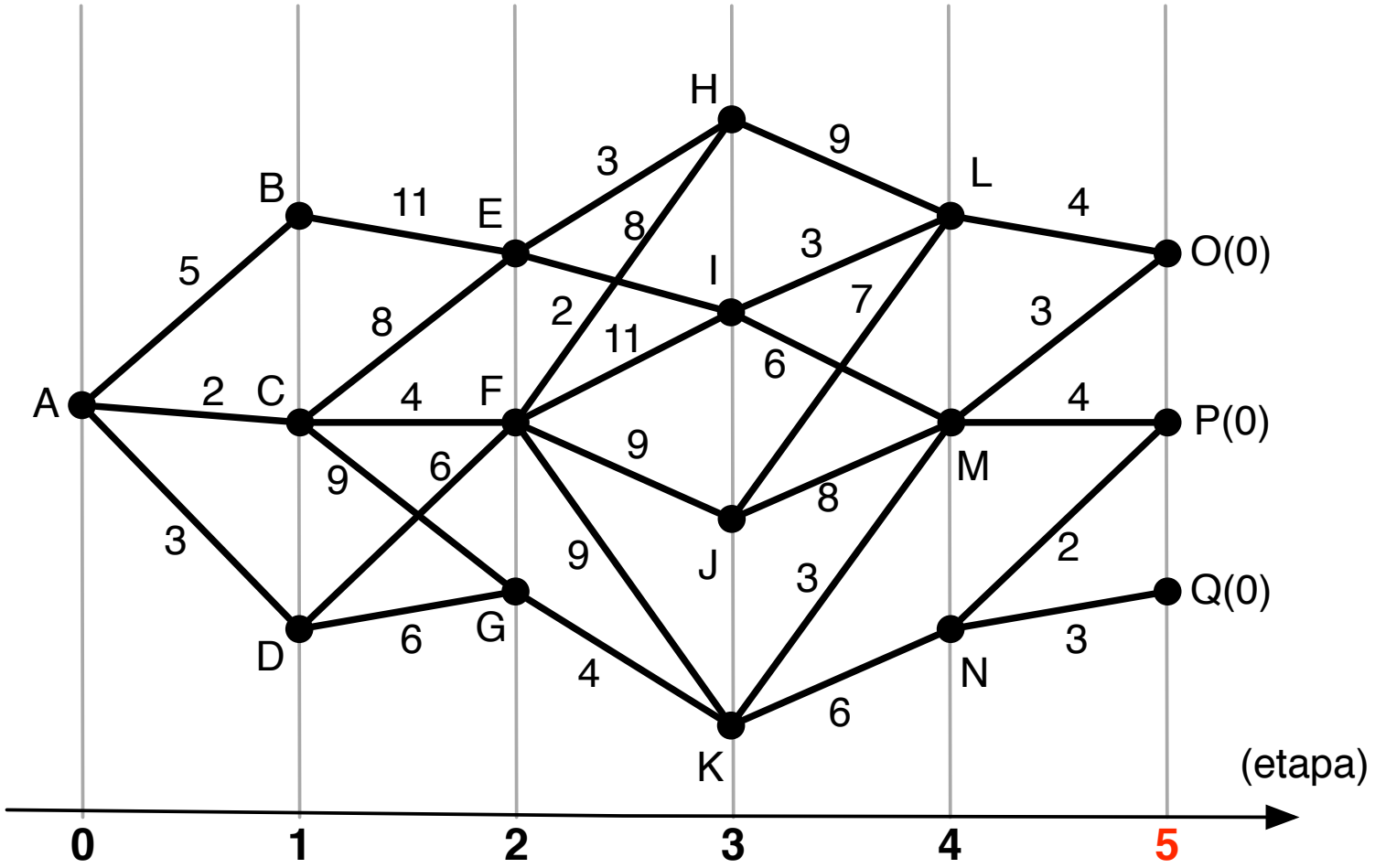
Método de Programación Dinámica (en optimización)

¿Como recorrer las etapas para llegar desde A hasta O, P, o Q (indistintamente) acumulando la menor distancia posible?



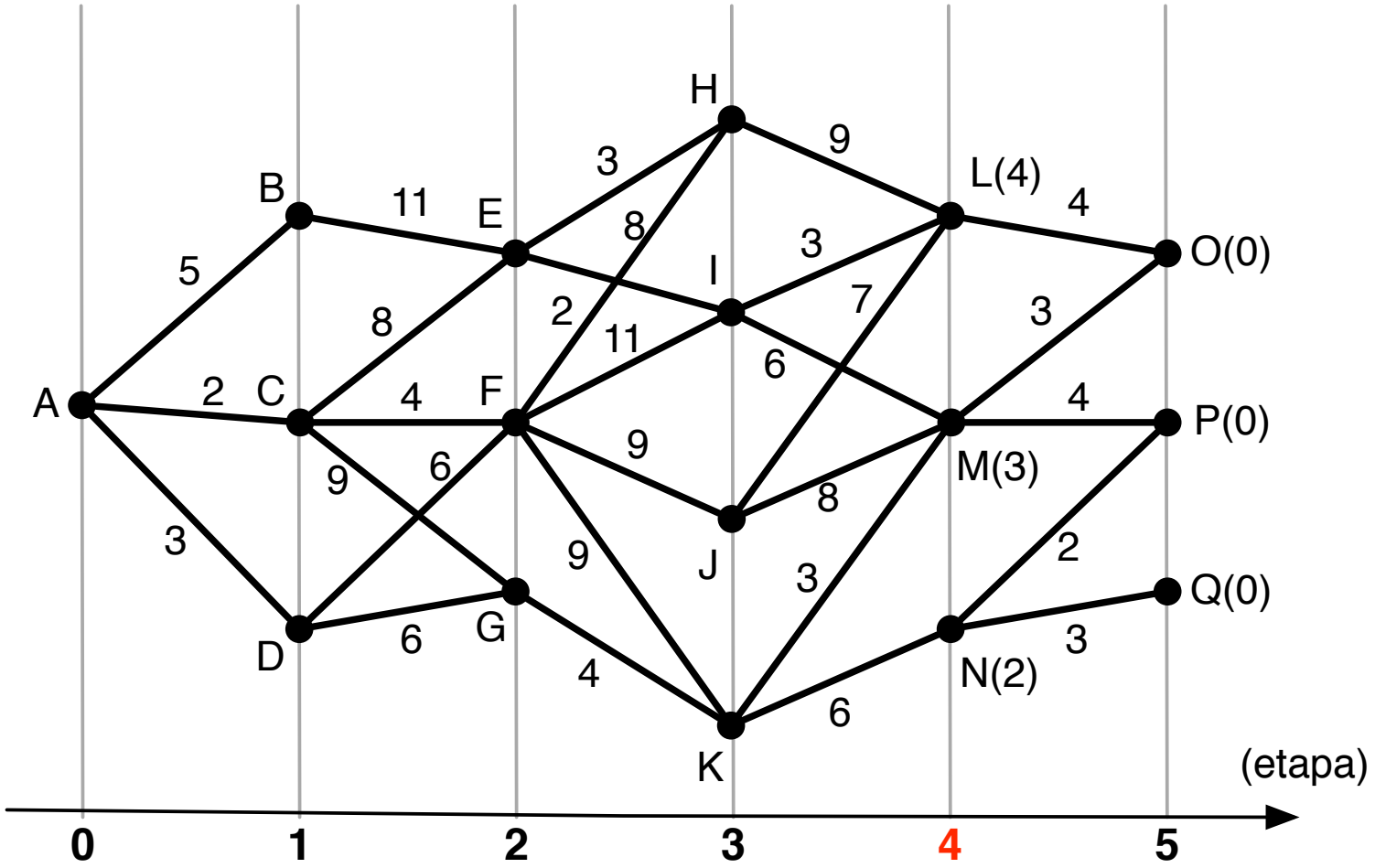
Método de Programación Dinámica (en optimización)

¿Como recorrer las etapas para llegar desde A hasta O, P, o Q (indistintamente) acumulando la menor distancia posible?



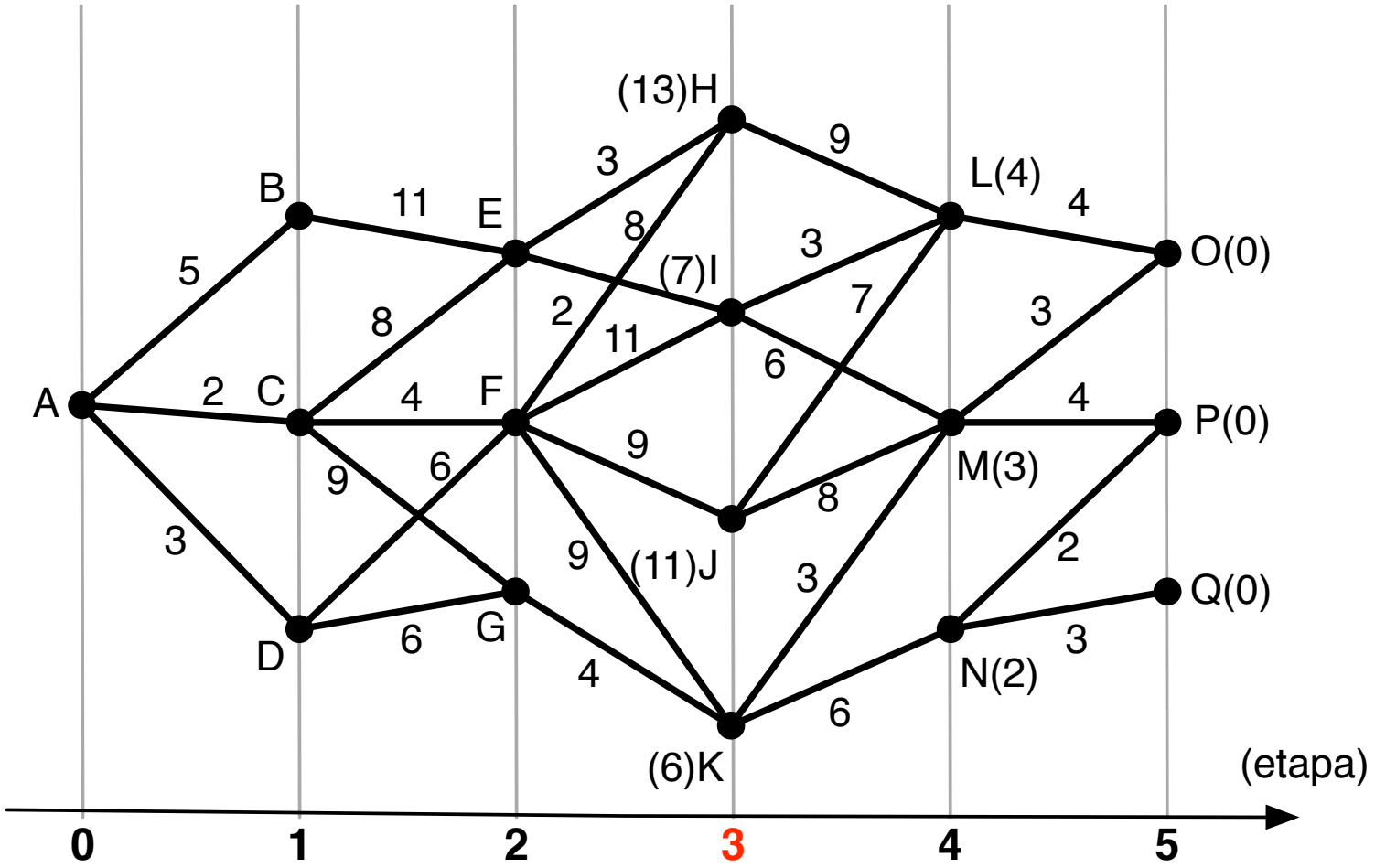
Método de Programación Dinámica (en optimización)

¿Como recorrer las etapas para llegar desde A hasta O, P, o Q (indistintamente) acumulando la menor distancia posible?



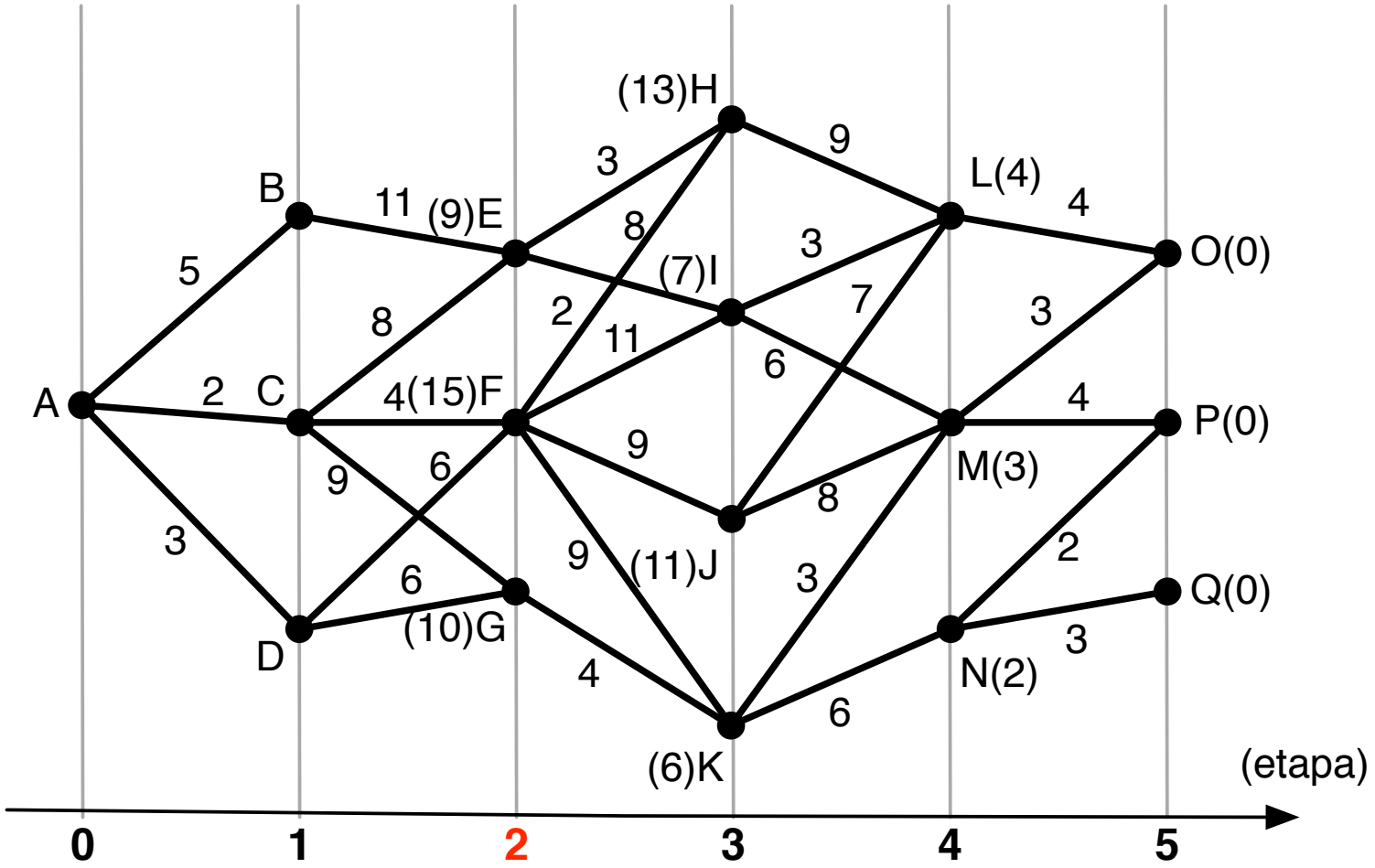
Método de Programación Dinámica (en optimización)

¿Como recorrer las etapas para llegar desde A hasta O, P, o Q (indistintamente) acumulando la menor distancia posible?



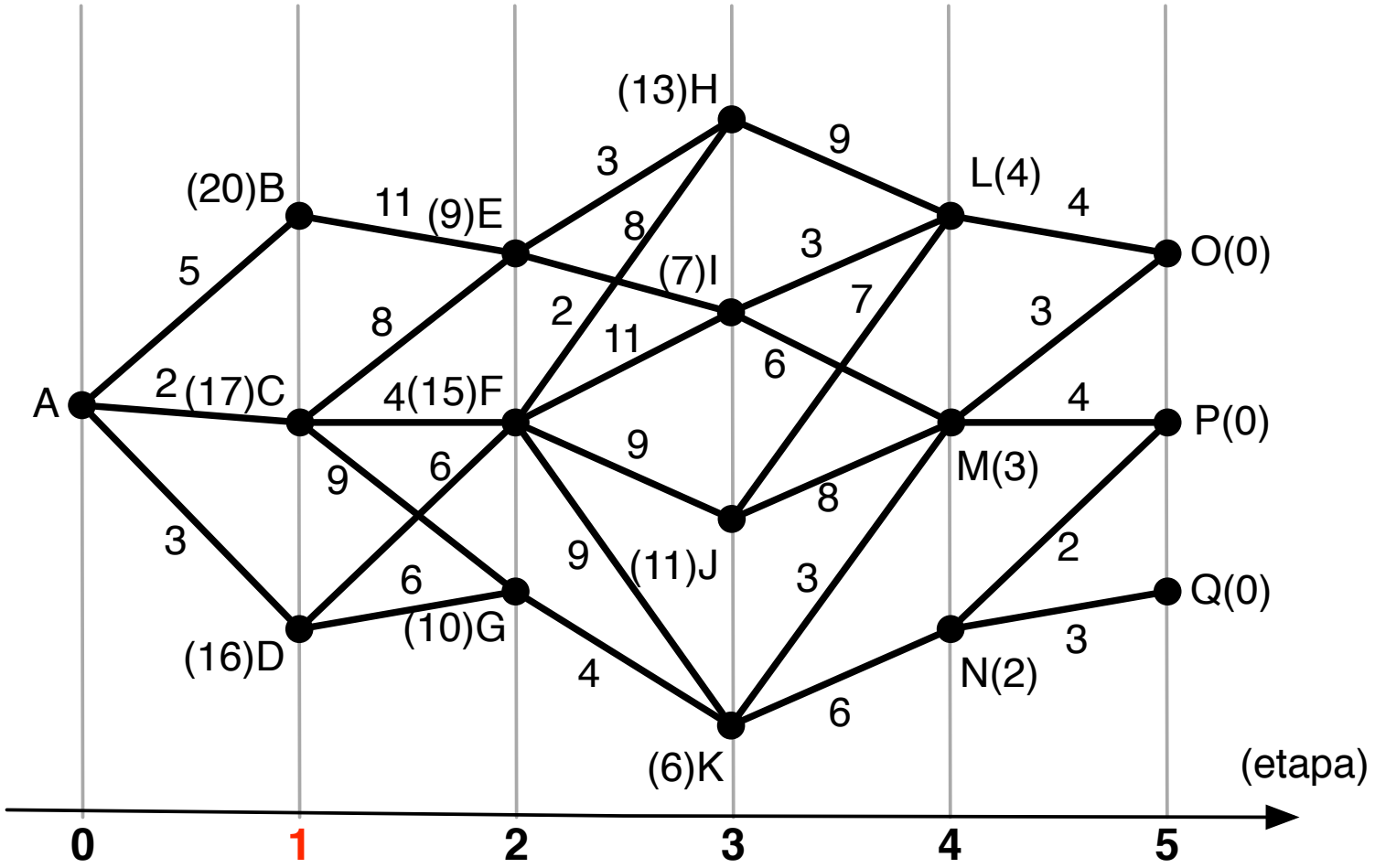
Método de Programación Dinámica (en optimización)

¿Como recorrer las etapas para llegar desde A hasta O, P, o Q (indistintamente) acumulando la menor distancia posible?



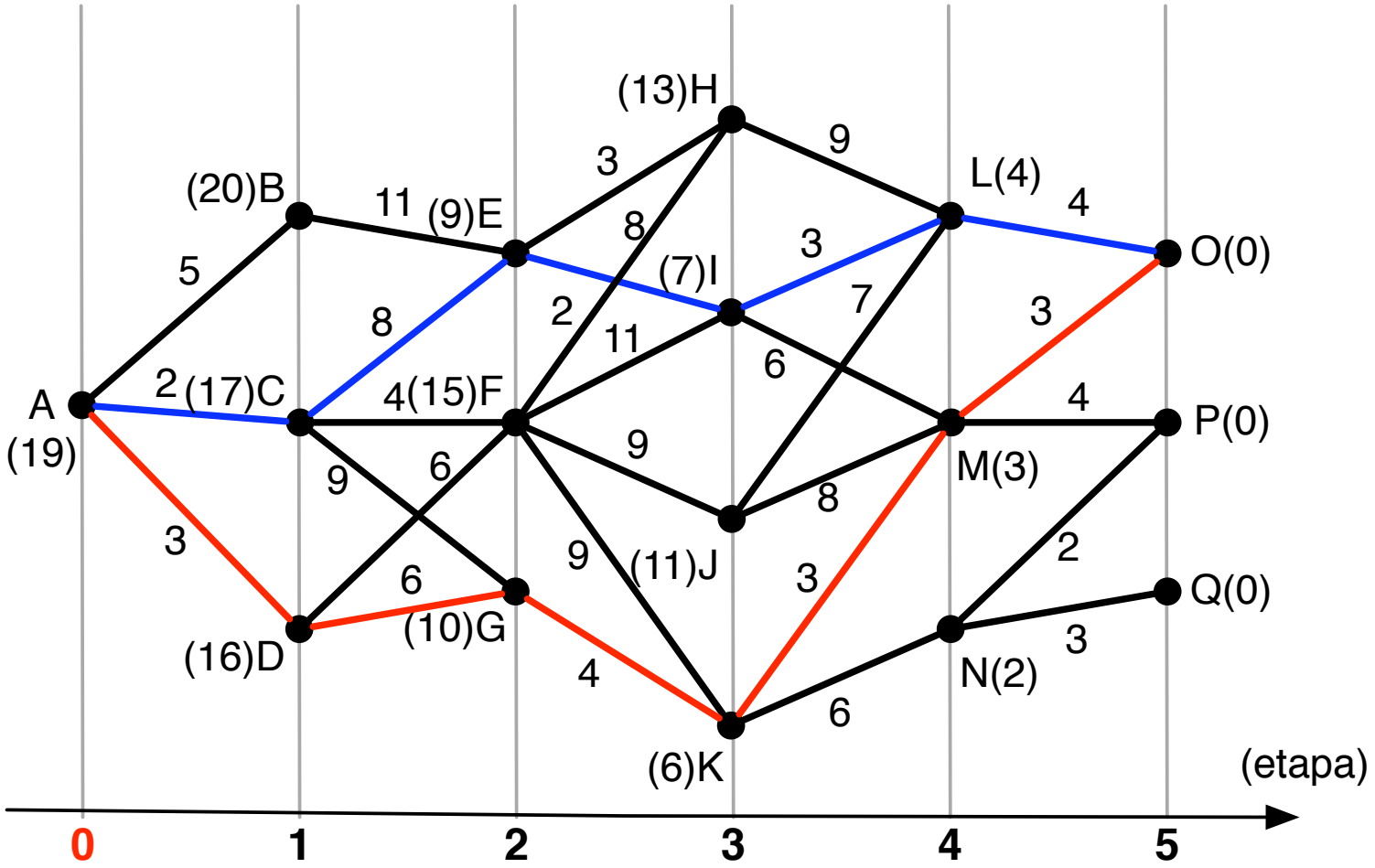
Método de Programación Dinámica (en optimización)

¿Como recorrer las etapas para llegar desde A hasta O, P, o Q (indistintamente) acumulando la menor distancia posible?



Método de Programación Dinámica (en optimización)

¿Como recorrer las etapas para llegar desde A hasta O, P, o Q (indistintamente) acumulando la menor distancia posible?



Algoritmo de Programación Dinámica

Suponemos que el problema consta de T etapas $(1, \ldots, T)$, que en cada etapa hay n estados $(1, \ldots, n$, los mismos por simplicidad) y que conocemos la función de costo final $f(i, T)$, con $i = 1, \ldots, n$.

Se conoce por último la función de costo $c(i, j, t+1)$, con el costo incurrido por pasar del estado i en t, al estado j en $t+1$.

La recursión queda: $f(i,t) = \min_{1 \leq j \leq n}\{c(i,j,t+1) + f(j,t+1)\}$. El valor óptimo es $f(i,1)$ para el(los) estado(s) inicial(es).

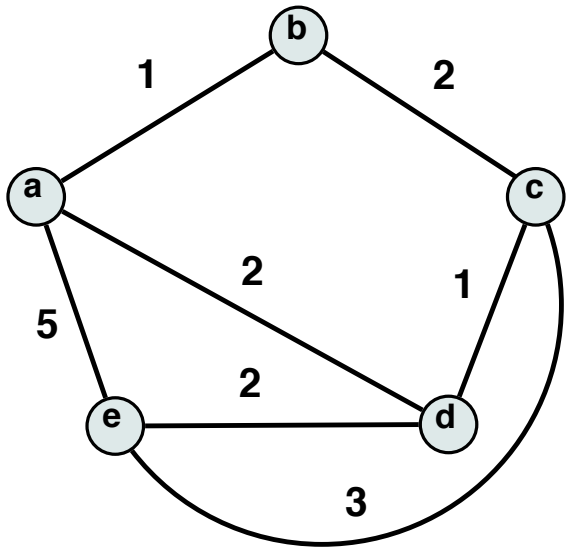
Es sumamente eficiente porque es del orden $O(Tn^2)$. Constituye una de las pocas excepciones de interés dentro de los problemas combinatorios, en las que hay algoritmos eficientes para resolverlo.

Enriqueciendo el número de estados se pueden modelar problemas más complejos (a costa de la eficiencia). Un caso importante es el de la Programación Dinámica Estocástica.

Contraejemplo a la Programación Dinámica

Aunque útil y eficiente, la programación dinámica carece de generalidad ante un problema combinatorio cualquiera. Se basa en una recursión greedy que requiere etapas, estados y cumplir además con el principio de optimalidad en esas etapas

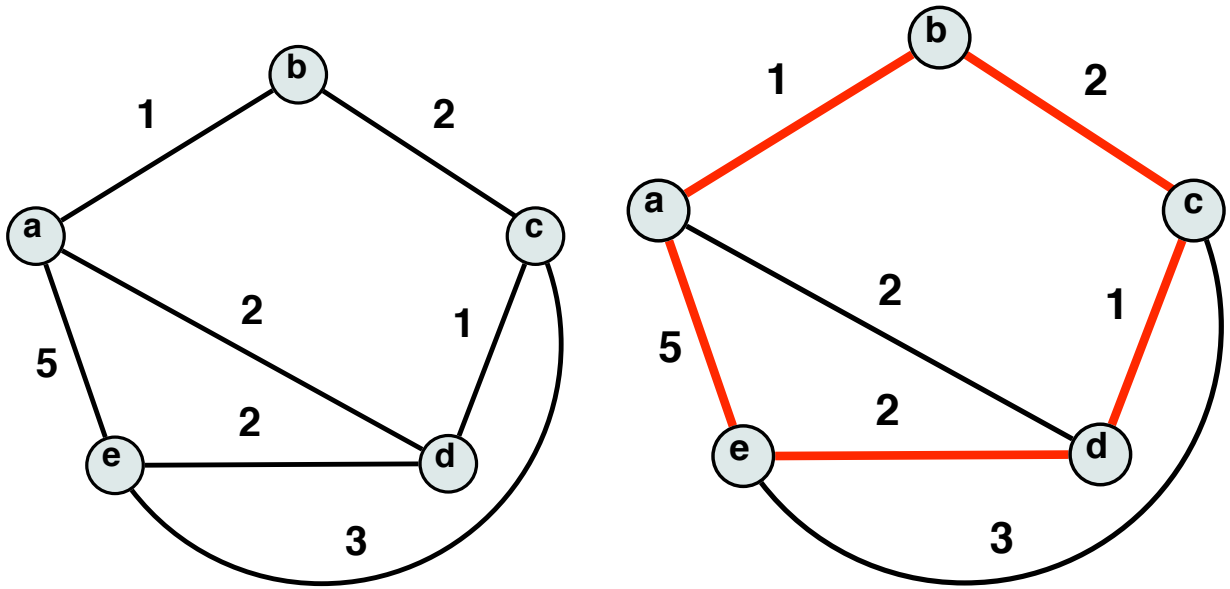
Los problemas *NP-Hard* no aceptan soluciones greedy. Uno de los ejemplos más clásicos es el TSP:



Contraejemplo a la Programación Dinámica

Aunque útil y eficiente, la programación dinámica carece de generalidad ante un problema combinatorio cualquiera. Se basa en una recursión greedy que requiere etapas, estados y cumplir además con el principio de optimalidad en esas etapas

Los problemas *NP-Hard* no aceptan soluciones greedy. Uno de los ejemplos más clásicos es el TSP:

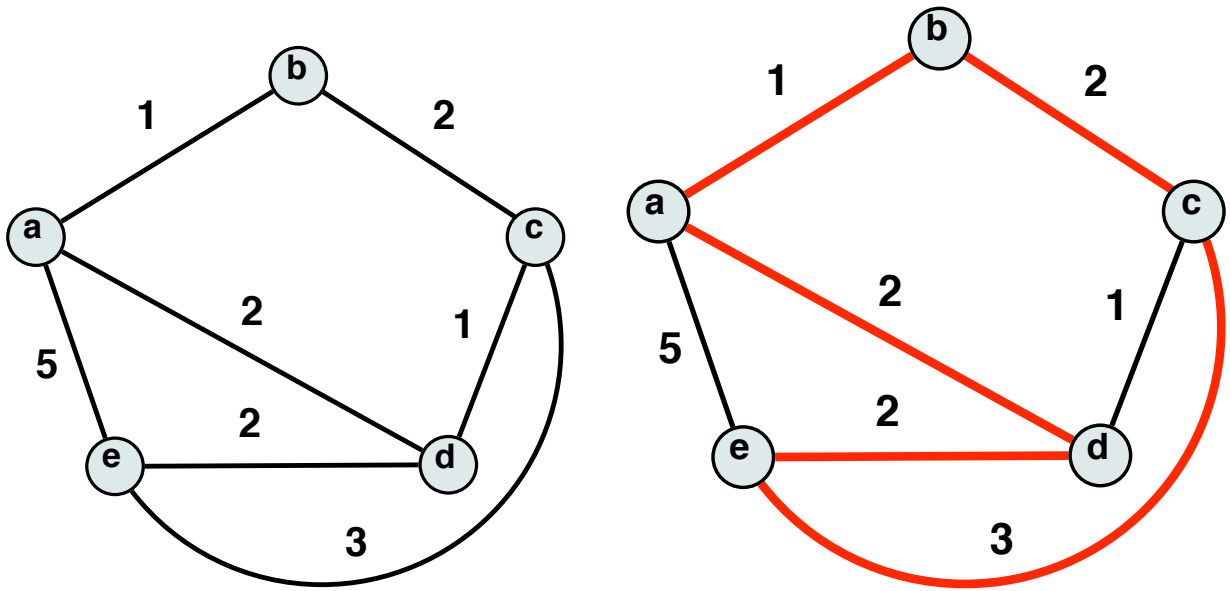


Solución greedy de costo 11.

Contraejemplo a la Programación Dinámica

Aunque útil y eficiente, la programación dinámica carece de generalidad ante un problema combinatorio cualquiera. Se basa en una recursión greedy que requiere etapas, estados y cumplir además con el principio de optimalidad en esas etapas

Los problemas *NP-Hard* no aceptan soluciones greedy. Uno de los ejemplos más clásicos es el TSP:



Solución óptima de costo 10.