.UBAfiuba
FACULTAD DE INGENIERÍA

Greedy: Presentación

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Algoritmos greedy

Metodología de resolución

de problemas de optimización (minimización o maximización)

Divide el problema en subproblemas

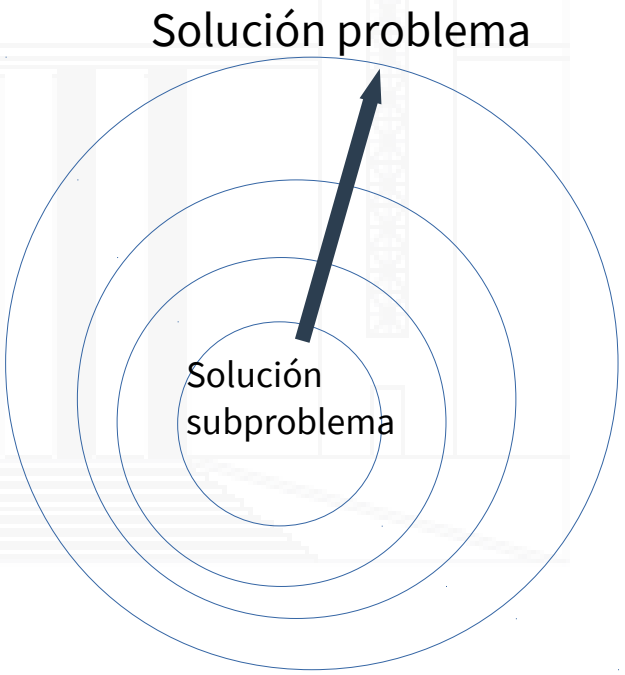
con una jerarquía entre ellos.

Cada subproblema

Se va resolviendo iterativamente

Mediante una elección heurística

Y habilita nuevos subproblemas



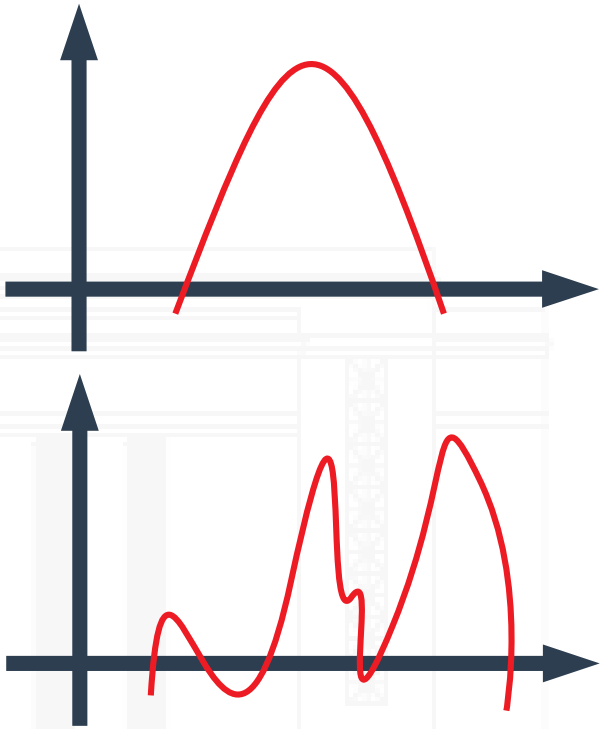
Problemas resolubles mediante greedy

Para cada problema

- Existen diferentes algoritmos greedy
- Algunos no resultan en la solución optima

No todos los problemas

- Pueden resolverse mediante un algoritmo greedy
- Deben tener ciertas propiedades



En el primer gráfico mediante la pendiente puedo iterativamente acercarme al valor máximo de la función
En el segundo caso ésta misma elección me puede llevar a un máximo local

Propiedades

Un problema debe contener las siguientes propiedades

para poder resolverse de forma optima mediante un algoritmo greedy

- **Elección greedy**
- **Subestructura óptima**

Elección Greedy

Seleccionar una solución óptima local

Esperando que la misma nos acerque a la solución optima global

La solución de un subproblema

Analiza el conjunto de los elementos del problema en el estado en que llegaron al mismo.

Elige heuristicamente la “mejor solución” local

Un subproblema

Está condicionado por las elecciones de los anteriores problemas

Condiciona a los subproblemas siguientes.

Subestructura óptima

Un problema

Contiene una subestructura óptima

Si la solución optima global del mismo

Contiene en su interior las soluciones optimas de sus subproblemas

La elección greedy

Iterativamente resolverá los subproblemas optimamente

Y nos llevará a la solución optima global

Lo sencillo y lo no tan sencillo...

Construir algoritmos greedy

Para resolver un problema es muy sencillo.

Es más complicado mostrar que el mismo

Resolverá óptimamente el problema

Tenemos que asegurarnos que

Para toda instancia del problema el algoritmo greedy nos provee la solución optima

Pasos en la construcción de una estrategia greedy

1. Determinar la subestructura óptima del problema
2. Construir una solución recursiva
3. Mostrar que si realizamos la elección greedy nos quedaremos en ultima instancia con solo 1 subproblema
4. Mostrar que la elección greedy se puede realizar siempre y de forma segura
5. construir el algoritmo recursivo que solucione el problema
6. Convertir el algoritmo recursivo en uno iterativo

.UBAfiuba

FACULTAD DE INGENIERÍA

Presentación realizada en Septiembre de 2020

.UBAfiuba
FACULTAD DE INGENIERÍA

Greedy: Interval Scheduling

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Interval Scheduling

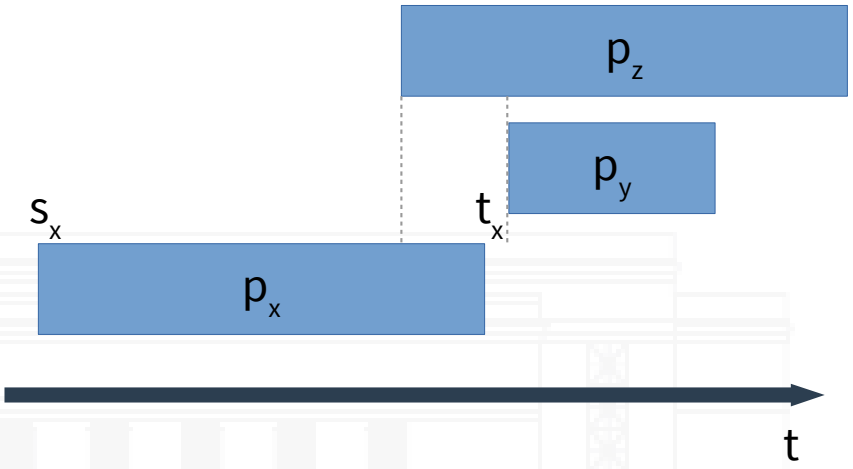
Sea

P un conjunto de n pedidos $\{p_1, p_2, \ldots, p_n\}$

Cada pedido i tiene

un tiempo si donde inicia

Un tiempo ti donde finaliza



Px y py son compatibles entre si. Sin embargo Pz no es compatible con ninguna de las anteriores

Un par de tareas $p_x, p_y \in P$

Son compatibles entre si, si - y solo si – no hay solapamiento en el tiempo entre ellas

Queremos

Seleccionar el subconjunto P de mayor tamaño posible de modo que todas las tareas seleccionadas sean compatibles entre si

Análisis del problema

Debemos determinar

Como seleccionamos los pedidos.

Al momento de seleccionar un pedido

Un subconjunto de pedidos serán incompatibles

Por lo tanto no elegibles en la solución

Iterativamente seleccionaremos pedidos

Mediante una heurística greedy

Y finalizaremos cuando no queden pedidos seleccionables

Selección greedy

Podemos construir

diferentes criterios a la hora de seleccionar los pedidos

Algunos de ellos

Aquel que comienza antes

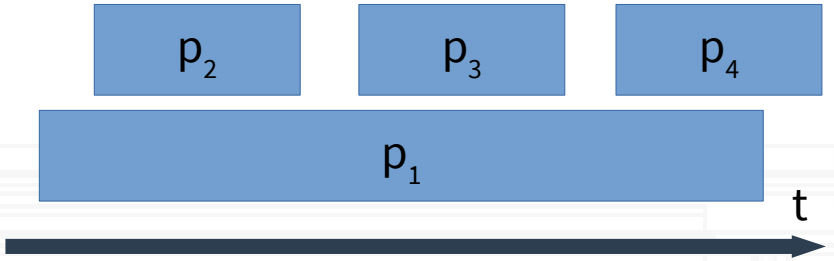
Aquel que dura menos tiempo

Aquel que tiene menos incompatibilidades

Aquel que termina antes

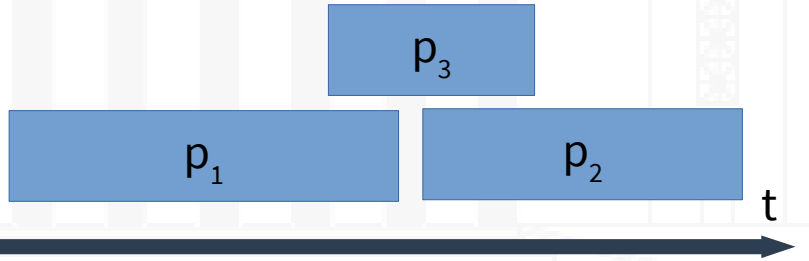
Heuristicas que no funcionan

Aquel que comienza antes



Con esta heurística seleccionaríamos P_1 y el óptimo es $\{p2, p_3, p_4\}$

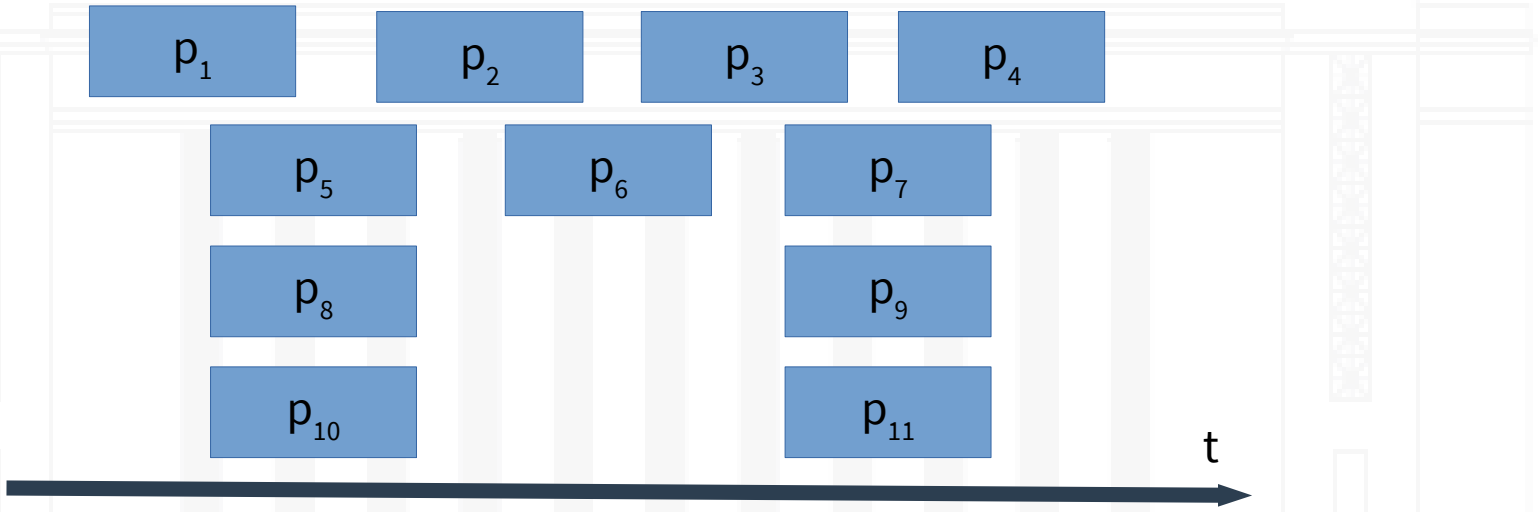
Aquel que dura menos tiempo



Con esta heurística seleccionaríamos P_3 y el óptimo es $\{p_1, p_2\}$

Heuristicas que no funcionan (cont.)

Aquel que tiene menos incompatibilidades



Con esta heurística seleccionaríamos primero P_6 y por lo tanto el tamaño máximo del conjunto final sera 3, el óptimo es $\{p_1, p_2, p_3, p_4\}$ con tamaño 4

Contraejemplos y demostración de optimalidad

En los ejemplos anteriores

Bastaba con encontrar un contraejemplo para desestimar la heuristica

A veces

puede ser difícil encontrar el contraejemplo

Si buscamos y no lo encontramos,

procedemos a intentar probar que utilizando ese criterio de selección, la misma nos lleva a obtener el optimo global

Heuristica: Aquel que termina antes

En este caso encontrar un contraejemplo

Aparenta ser imposible

Por lo tanto intentaremos

Demostrar que usando esa heurística corresponde a la eleccion greedy correcta.

Si demostramos logicamente que el resultado es optimo, entonces tendremos nuestro algoritmo greedy

Describamos como funcionará

este algoritmo

Pseudocódigo

```
Sea P set de pedidos
Sea A subconjunto maximo

Mientras P <> vacio

    Sea i el pedido en P con menor tiempo de finalización

    A = A + {i}

    Quitar de P todos los pedidos incomplatibles con i

Retornar A
```

Análisis del algoritmo: Set compatible

El set retornado por el algoritmo

Es compatible

Esto se obtiene por la naturaleza misma del algoritmo:

En cada iteración se eliminan todos los pedidos incompatibles del seleccionado.

… queda ver si ademas es óptimo

Análisis del algoritmo: Optimalidad

Llamaremos

O a un set optimo

No podemos asegurar que

A=O (podrían existir varias selecciones diferentes de tareas con la misma cantidad de pedidos)

Queremos ver que

|A|=|O|

Para demostrar esto

Utilizaremos la idea de que el algoritmo greedy “se mantiene por delante” de la solución Optima

Optimalidad - Idea

Podemos enumerar los elementos de A

Según el orden en el que fueron seleccionadas $\{i_1, i_2, \ldots i_k\}$ con $|A|=k$

Ademas, podemos enumerar los elementos de O

Ordenados por fecha de inicialización del pedido $\{j_1, j_2, \ldots, j_m\}$ con $|O|=m$

(como los pedidos son compatibles, entonces es lo mismo ordenar por fecha de inicialización o finalización)

Compararemos las soluciones parciales construidas por greedy

Con los segmentos iniciales de la solución greedy

Mostraremos que greedy

“lo está haciendo mejor” de una forma paso a paso. (tiene igual o mas pedidos seleccionados)

Optimalidad: 1er pedido

Analizamos

el primer elemento de O y A

Por como se selecciona en greedy

podemos ver que: $f(i_1) \leq f(j_1)$

Pueden ser el mismo pedido (o dos pedidos que finalizan al mismo tiempo)

En ese caso $f(i_1) = f(j_1)$

Pero nunca puede ser $f(i_1) > f(j_1)$

Seria una violación a la forma de seleccionar en el greedy

Optimalidad: Inducción

Queremos demostrar que

Para todo r ≤ k se cumple que $f(i_r) \leq f(j_k)$

Utilizaremos inducción para probarlo

Para el caso base r=1 esto es cierto

Asumiremos que es cierto para r-1 con r>1 (hipótesis inductiva)

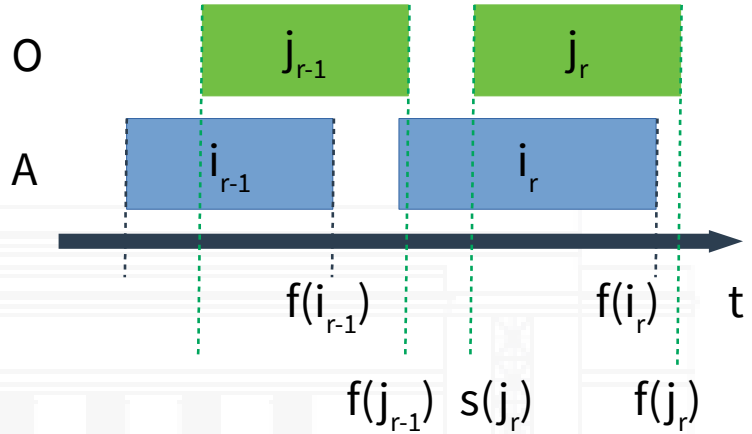
Por lo tanto asumimos $f(i_{r-1}) \leq f(j_{r-1})$

Como O esta compuesto por pedidos compatibles

$f(j_{r-1}) \leq s(j_r)$

Con estas 2 inecuaciones podemos obtener

$f(i_{r-1}) \leq s(j_r)$



Optimalidad: Inducción (cont.)

La inecuación

$f(i_{r-1}) \leq s(j_r)$

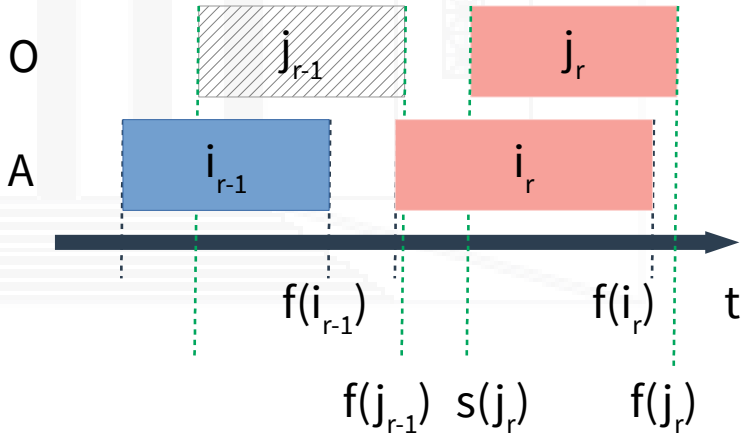
Implica que en el momento que greedy seleccionó i_r

También estaba disponible j_r

Greedy selecciona aquel disponible que termine antes

Entonces $f(i_r) \leq f(j_r)$

Lo que concluye el paso inductivo



Optimalidad: Colorario

Podemos afirmar (en base la demostración inductiva) que el algoritmo greedy retorna un set A óptimo

Si A no es optimo entonces el O debe tener mas pedidos (m > k), |O|=m y |A|=k

Dado que para todo r ≤ k se cumple que $f(i_r) \leq f(j_k)$

Si r=k entonces $f(i_k) \leq f(j_k)$

Como m > k,

Implica que existe en O un pedido j_{k+1} que comienza luego que j_k termina

Demostramos que si j_{k+1} finaliza luego que j_k

Entonces finaliza luego que i_k

Lo que indicaría que greedy, luego de seleccionar i_k

Aun tendría pedidos compatibles para seleccionar y no habría seleccionado ninguno

Lo que seria una contradicción

Implementación

Para implementar de forma eficiente el psudocodigo

Debemos encontrar la forma de recorrer los pedidos de forma conveniente según nuestra heuristica

Se puede ordenar los pedidos por tiempo de finalización

De esa forma elegimos el primero disponible

Luego iteramos

ignorando los incompatibles (comparando la fecha de finalización del último seleccionado con la fecha de inicio del analizado)

Y seleccionando el próximo disponible (pasando a ser el ultimo seleccionado)

Complejidad

El proceso de ordenamiento

Es O(n log n)

La iteración

Es O(n)

Por lo tanto la complejidad temporal total

Es O(n log n)

La complejidad espacial

Es O(n) ← si todos los pedidos son compatibles elijo los n

```
Sea P set de pedidos
Sea A subconjunto maximo

Ordenar P por fecha de finalización

A = A + {P[1]}
finilizacion = P[1].finalizacion
Desde i=2 a n

     Si P[i].inicio >= finalizacion
          A = A + {P[i]}
          finalizacion = P[i].finalizacion


Retornar A
```

Ejemplo

Sean los siguientes pedidos

i	s	f
1	4	6
2	2	5
3	1	3
4	6	8
5	3	7

Los ordenamos por tiempo f

i	s	f
3	1	3
2	2	5
1	4	6
5	3	7
4	6	8

Seleccionamos algorítmicamente

i	s	f	A	fin
3	1	3	si	3
2	2	5	X	3
1	4	6	si	6
5	3	7	X	6
4	6	8	si	8

Siendo la solución optima

{1,3,4}

.UBAfiuba

FACULTAD DE INGENIERÍA

Presentación realizada en Septiembre de 2020

.UBAfiuba
FACULTAD DE INGENIERÍA

Interval Partitioning Problem

Teoría de Algoritmos I (75.29 / 95.06)
Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Problema

Sea

E un conjunto de n eventos $\{e_1, e_2, \ldots, e_n\}$

S un conjunto de salas

Cada evento e tiene (un intervalo)

una fecha f_s de inicio

Una fecha f_e de finalización

Queremos

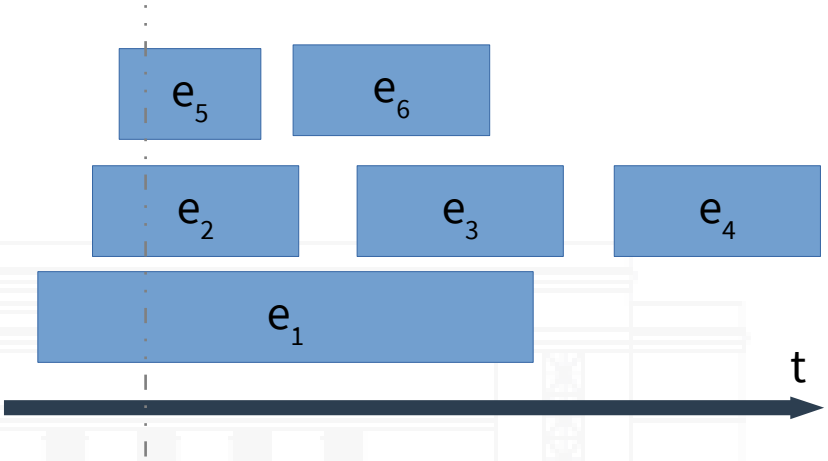
Asignar cada evento a una sala minimizando la cantidad de salas utilizadas.

(dos eventos no pueden compartir tiempo simultaneo en una misma sala)

Análisis del problema

Debemos determinar

Como asignar los eventos a las salas.



Al momento de asignar una evento

Un subconjunto de eventos podrán ser incompatibles para esa misma sala

Por lo tanto al menos se requerirá una sala extra

El número máximo de eventos incompatibles entre si

Determina el número máximo de salas necesarias

Llamaremos profundidad a este parámetro

Una primera solución (problemática)...

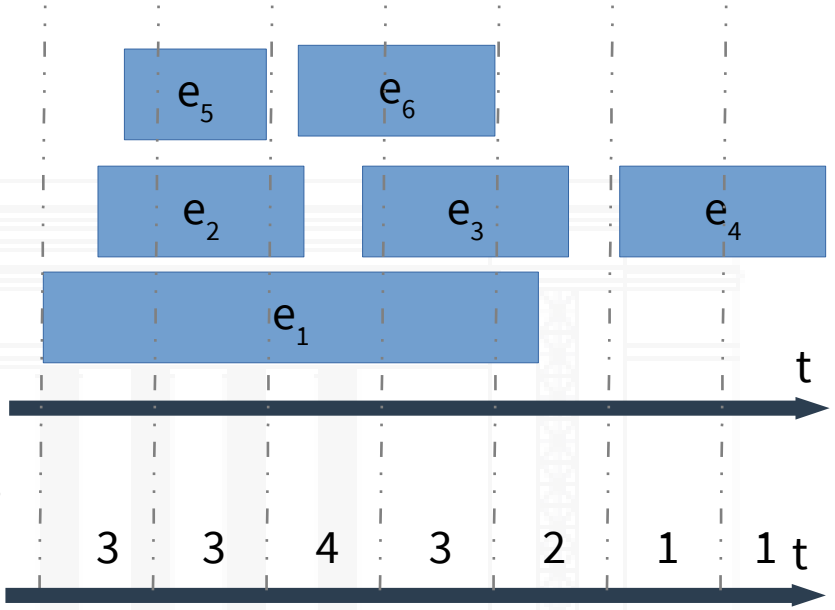
Discretizar en intervalos el uso de las salas

Por cada intervalo llevar un contador de eventos

Luego verificar que intervalo tiene mas eventos y ese será el número de salas

No cualquier separación de intervalos sirve

Una elección incorrecto llevará a resultados erroneos



El número de intervalos i requeridos puede ser muy grande

La complejidad temporal y espacial del algoritmo queda en su función → O(i*e)

No resuelve el problema de asignación

Solo de determinación de la cantidad de salas

Elaborando una solución eficiente...

El número máximo de salas posible

Corresponde al número total de eventos

(en el caso extremo todos los eventos coinciden en algún momento)

Queremos una solución

Que requiera como recursos una cantidad similar a la profundidad del los eventos

Que requiera recorrer la menor cantidad posible a los eventos

Que sea de tipo greedy

Diseño de una solución óptima

Sea d la profundidad del conjunto de eventos

Utilizaremos las etiquetas {1,2,..,d} para asignar eventos a una sala

Ordenaremos los eventos

por su fecha de inició → O(nlogn)

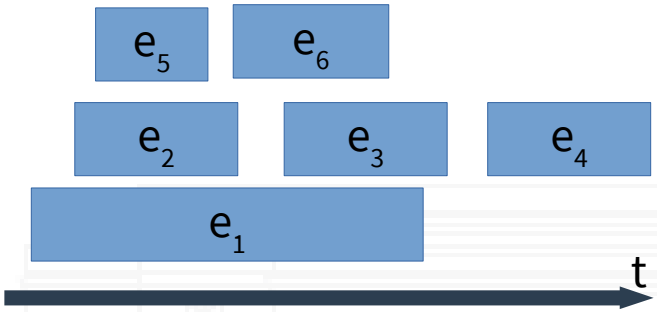
Iteraremos por los eventos ordenados

Etiquetando uno a uno con una etiqueta que no haya sido asignado previamente a un evento que se superpone

Pseudocódigo

```
Sea E set de eventos
Ordenar E los eventos por fecha de inicio

Por cada evento Ej
     Por cada evento Ei que precede a Ej y se superpone con Ej
          Excluir la etiqueta de Ei para Ej

     Si existe una etiqueta {1,2,…,d} que no fue excluida
          asignar la menor a Ej
     Sino
          dejar Ej sin etiquetar

Retornar los eventos etiquetados
```

Ejemplo



Eventos ordenados: e1, e2, e5, e6, e3, e4

Evento	Superponen	excluidas	etiqueta
e1	-	-	1
e2	e1	1	2
e5	e1,e2	1,2	3
e6	e1,e2	1,2	3
e3	e1,e6	1,3	2
e4	-	-	1

```
Sea E set de eventos
Ordenar E los eventos por fecha de inicio

Por cada evento Ej
     Por cada evento Ei que precede a Ej y se
                                   superpone con Ej

          Excluir la etiqueta de Ei para Ej

     Si existe una etiqueta {1,2,…,d} que no
                                   fue excluida

          asignar la menor a Ej

     Sino

          dejar Ej sin etiquetar

Retornar los eventos etiquetados
```

Complejidad

Ordenar

O(nlogn)

Iterar por cada evento

O(n)

Verificar por cada evento anterior

O(n)

Complejidad total

$O(n^2)$

```
Sea E set de eventos
Ordenar E los eventos por fecha de inicio

Por cada evento Ej
     Por cada evento Ei que precede a Ej y se
                                  superpone con Ej

          Excluir la etiqueta de Ei para Ej

     Si existe una etiqueta {1,2,…,d} que no
                                     fue excluida

          asignar la menor a Ej

     Sino

          dejar Ej sin etiquetar

Retornar los eventos etiquetados
```

Optimalidad

¿Puede quedar un evento sin etiquetar?

Sabemos que tenemos una profundidad de d (con d≤n)

Al analizar el evento j. Como mucho d-1 eventos anteriores se superponen

Entonces podemos etiquetar en el peor de los casos el evento con “d”

¿Pueden recibir 2 eventos superpuestos la misma etiqueta?

No, por diseño del algoritmo.

¿Puedo asignar una etiqueta no necesaria?

No, siempre se reutilizan las etiquetas a medida que se van liberando

¿Podemos mejorar la eficiencia?

Utilizando estructura de datos diferentes

Podemos evitar tener la segunda iteración de eventos de exclusión.

Nos interesa conocer la primera etiqueta disponible

Proponemos almacenar por etiqueta utilizada la fecha de liberación

(la fecha de finalización del evento última que tiene asignada)

Al evaluar un nuevo evento (con su respectiva fecha de inicio)

Queremos saber cual es la primera etiqueta (sala) que se libera.

Si la fecha de inicio del evento es anterior a la fecha de liberación de esta sala necesitaremos una sala nueva.

Si no, la asignamos en esa sala y actualizamos la fecha de liberación de esta sala

Utilizaremos

una cola de prioridades (ej: heap) de mínimos.

Pseudocódigo (método mejorado)

```
Ordenar E los eventos por fecha de inicio = {e1, e2, … en }
Q cola de prioridad
u=1 // ultima sala utilizada sala

ev = E[1] //tomo el primer evento
ev.etiqueta = u //etiqueto el evento en la primera sala

Q.push({ev.finalizacion, u})  // sala con fecha de liberacion y etiqueta

Desde j=2 a n
     ev = E[j]
     s = Q.min()

     si ev.inicio >= s.liberacion   // puedo asignar el evento en una sala existente
          s = Q.pop()
          ev.etiqueta = s.etiqueta
          Q.push({ev.finalizacion}, s.etiqueta)
     sino
          u=u+1
          ev.etiqueta = u
          Q.push({ev.finalizacion}, u)

Retornar los eventos etiquetados
```

Complejidad

Ordenar

O(nlogn)

Utilizando un heap

Push,pop → O(log(n))

Min → O(1)

Iteramos 1 vez por evento → O(n)

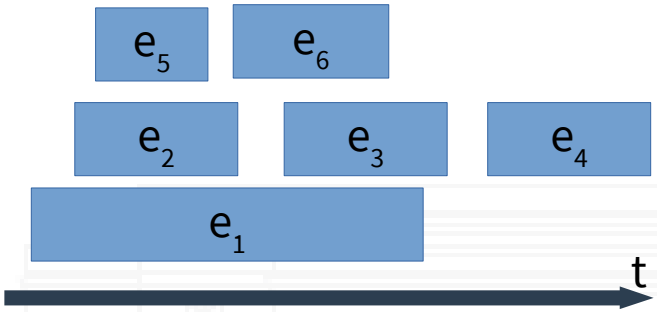
Por cada evento realizamos operaciones en el heap

Complejidad total

O(nlogn)

```
Ordenar E los eventos por fecha de inicio
Q cola de prioridad
u=1 // ultima sala utilizada sala

ev = E[1]
ev.etiqueta = u

Q.push({ev.finalizacion, u})

Desde j=2 a n
     ev = E[j]
     s = Q.min()

     si ev.inicio >= s.liberacion
          s = Q.pop()
          ev.etiqueta = s.etiqueta
          Q.push({ev.finalizacion},
s.etiqueta)
     sino
          u=u+1
          ev.etiqueta = u
          Q.push({ev.finalizacion}, u)

Retornar los eventos etiquetados
```

Ejemplo



Eventos ordenados: e1, e2, e5, e6, e3, e4

Evento	Heap	etiqueta
e1	{e1,1}	1
e2	{e2,2}{e1,1}	2
e5	{e5,3}{e2,2}{e1,1}	3
e6	{e2,2}{e1,1}{e6,3}	3
e3	{e1,1}{e3,2}{e6,3}	2
e4	{e3,2}{e6,3}{e4,1}	1

```
Ordenar E los eventos por fecha de inicio
Q cola de prioridad
u=1 // ultima sala utilizada sala

ev = E[1]
ev.etiqueta = u

Q.push({ev.finalizacion, u})

Desde j=2 a n
     ev = E[j]
     s = Q.min()

     si ev.inicio >= s.liberacion
          s = Q.pop()
          ev.etiqueta = s.etiqueta
          Q.push({ev.finalizacion},
s.etiqueta)
     sino
          u=u+1
          ev.etiqueta = u
          Q.push({ev.finalizacion}, u)

Retornar los eventos etiquetados
```

.UBAfiuba

FACULTAD DE INGENIERÍA

Presentación realizada en Abril de 2021

.UBAfiuba
FACULTAD DE INGENIERÍA

Greedy: Planificación para minimizar latencia

Teoría de Algoritmos I (75.29 / 95.06)
Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Planificación para minimizar el latencia (retraso)

Sea

P un conjunto de n pedidos $\{p_1, p_2, \ldots, p_n\}$

Cada pedido i tiene

Una duración ti (no fraccionable)

Una fecha de entrega (deadline) di

Queremos

programar el inicio de cada unos de los pedidos a realizar sin superposición de forma de minimizar la máxima latencia

Latencia

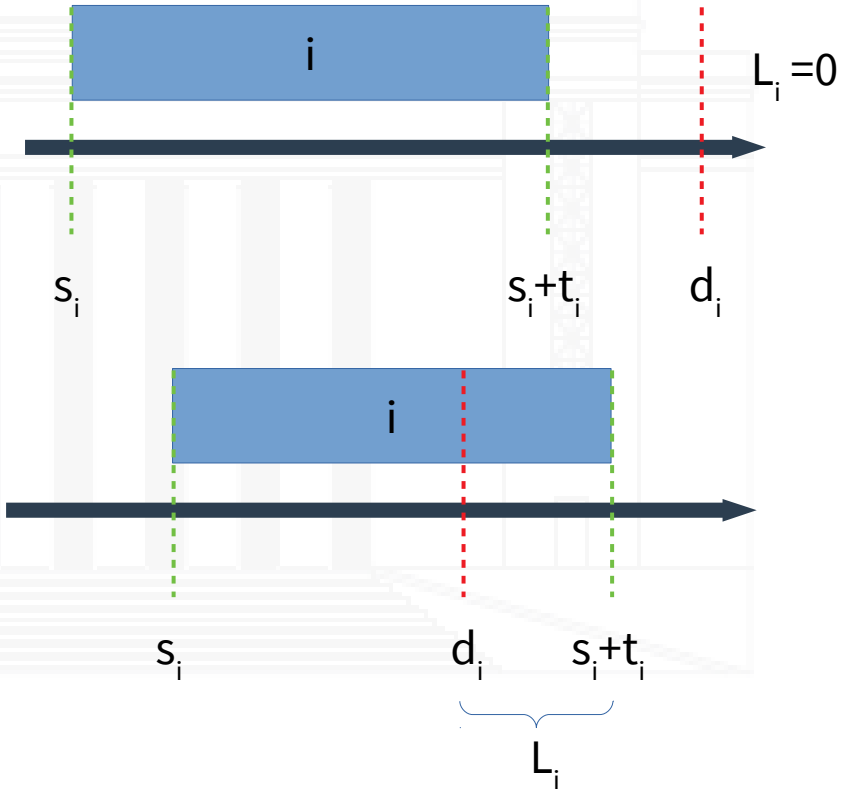
Si la tarea $i=(t_i,d_i)$

Se programa para comenzar en el tiempo s_i

Diremos que su latencia

$l_i = s_i + t_i - d_i$ si $(s_i + t_i - d_i) \geq 0$

$l_i = 0$ si $(s_i + t_i - d_i) < 0$



La latencia máxima

de una secuencia de pedidos programados

Corresponde a la maxima de las latencias

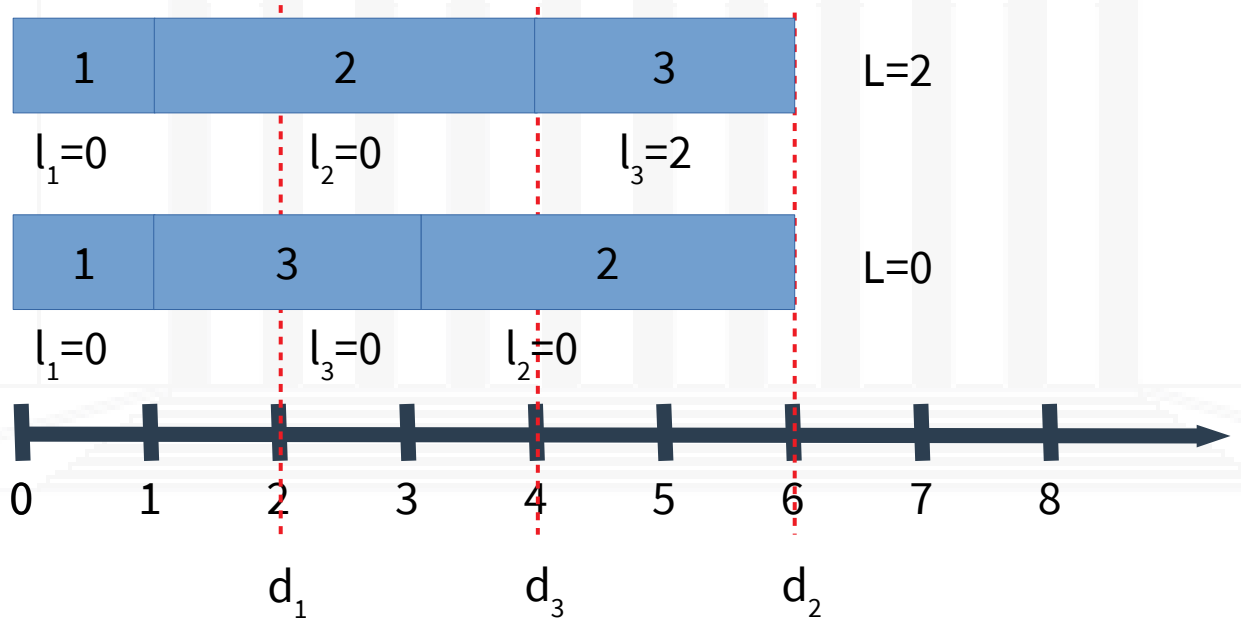
$L = Max_{1 a n}\ l_x$

Ejemplo

Contamos con los siguientes pedidos

1=(1,2) 2=(3,6) 3=(2,4)

¿Cómo los programamos?



Estrategia greedy

Podemos procesar los pedidos

De acuerdo a la duración de cada uno (del mas corto al más largo o viceversa)

De acuerdo a la diferencia entre el deadline y la duración ($d_i - t_i$)

De acuerdo al deadline (del más cercano al mas lejano)

Los primeros 2 no sirven

Podemos rápidamente encontrar contraejemplos que los desacredite

El tercero parece funcionar

Debemos demostrarlo!

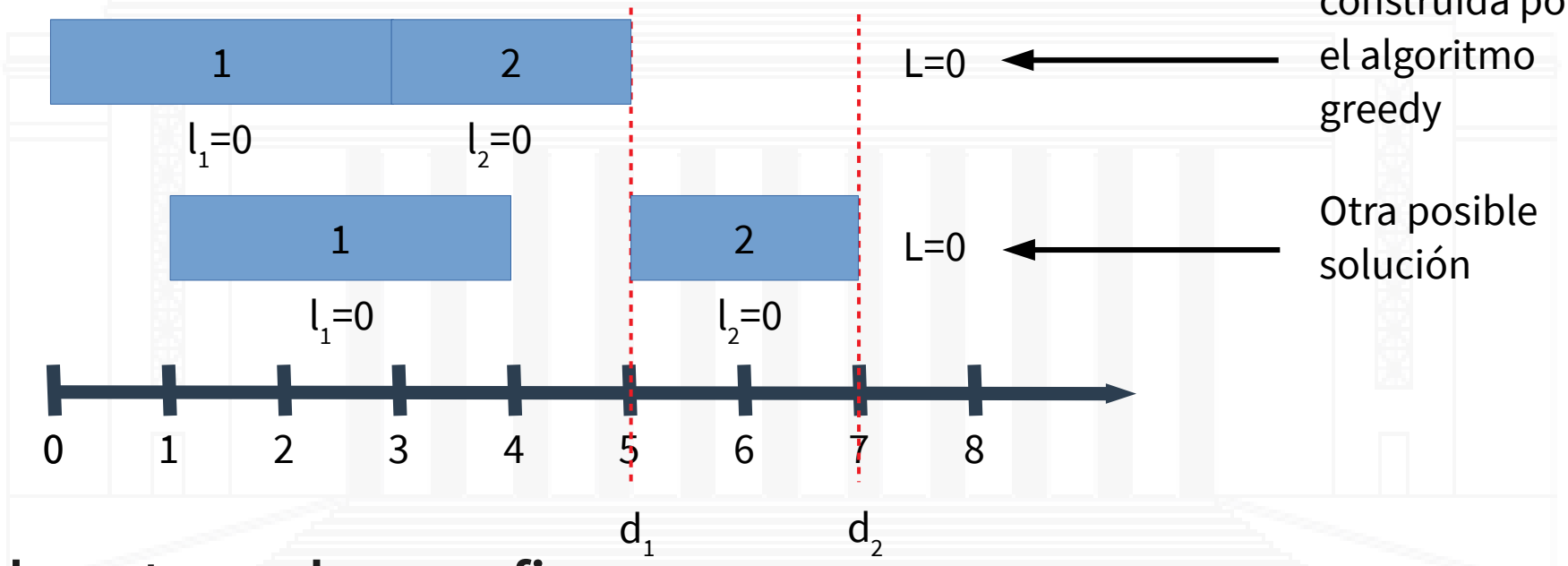
Pseudocódigo

```
Ordenar los pedidos en función de su deadline
(Asumiremos por simplicidad la notación d1 ≤ d2 ≤ … ≤ dn)

Inicialmente f=s    // s es la hora de inicio (podemos suponerlo como 0)
Desde el pedido i=1 al n
    Asignar el pedido i al intervalo s(i)=f a f(i)=f+ti
    f = f + ti

Retornar el set de intervalos programados [s(i), f(i)] para i=1 a n
```

Análisis de la solución: Tiempos muertos

La solución A que construye el algoritmo greedy

No deja tiempo "muertos" o de inactividad



Igualmente, podemos afirmar que

Existe una solución optima sin tiempos muertos

Optimalidad de la solución

Consideraremos una solución optima O

Esta puede ser cualquier solución optima posible

Intentaremos transformarla paso a paso

sin perder la optimalidad

Llegando finalmente a una programación

Idéntica a “A” la conseguida por el algoritmo greedy

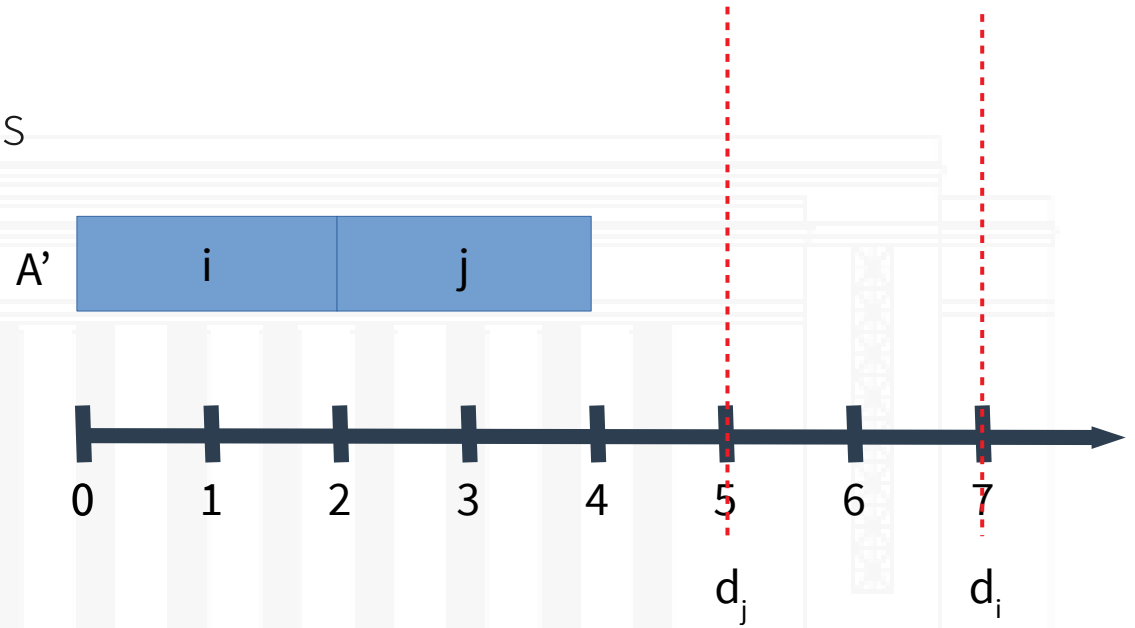
Llamaremos a este método de probar la optimalidad

Como un “argumento de intercambio”

Inversiones

Sea

A’ una programación de pedidos



Diremos

Que A’ tiene una <u>inversión</u>

Si

Un pedido $i=(t_i,d_i)$ esta programado antes de otro pedido $j=(t_j,d_j)$ y $d_j < d_i$

Programación sin inversión ni tiempos muertos

Sean

B,C programaciones sin inversiones ni tiempos muertos.

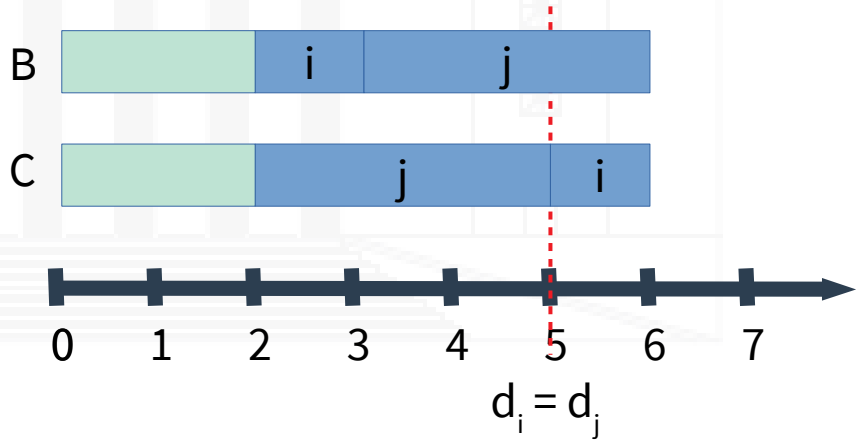
Entonces

B y C tienen el mismo tiempo de latencia total

Si B y C no tienen inversiones pero son diferentes

Entonces existen pedidos que comparten el mismo deadline

Invertir el orden de esos pedidos no altera la latencia máxima (*1)



Demostración: Paso 1

Sea

O una programación optima

Si O tiene tiempos muertos

Podemos eliminarlos desplazando los pedidos sin dejar de ser optimo

Si O tiene inversiones

Entonces existen al menos 2 pedidos contiguos i,i+1 tal que $d_{i+1} < d_i$

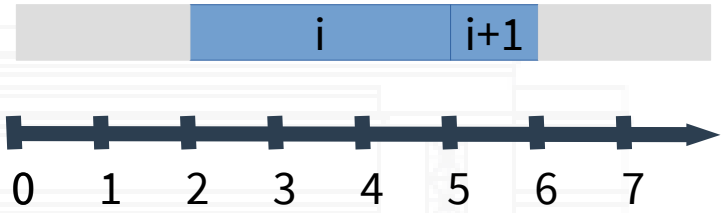
Podemos intercambiarlas para tener una inversión menos

Demostración: Inversión

Sea

O una programación optima con inversiones

i,i+1 una inversión de pedidos en O $(d_{i+1} < d_i)$



Realizamos una inversión entre i e i+1

Sabemos que la latencia máxima no puede disminuir (O no sería óptimo)

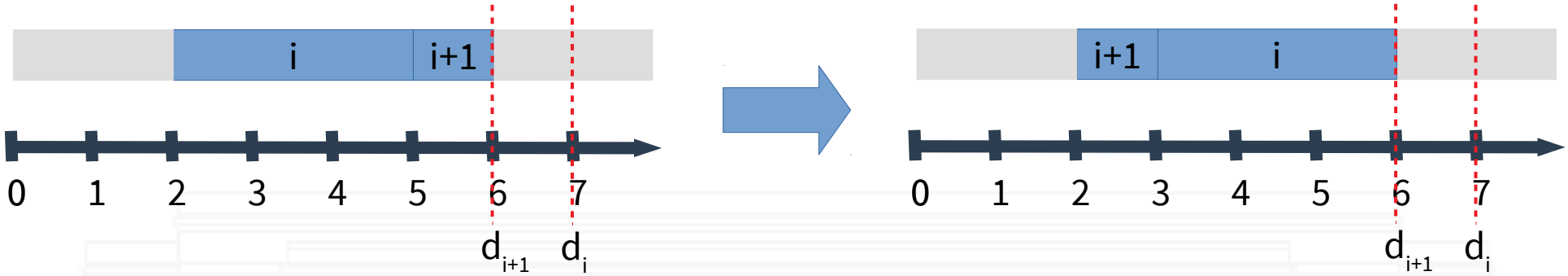
La latencia de i+1 puede mantenerse o disminuir

La latencia de i puede aumentar o mantenerse

La cantidad de inversiones disminuye en 1

No modifica la latencia de los pedidos no involucrados

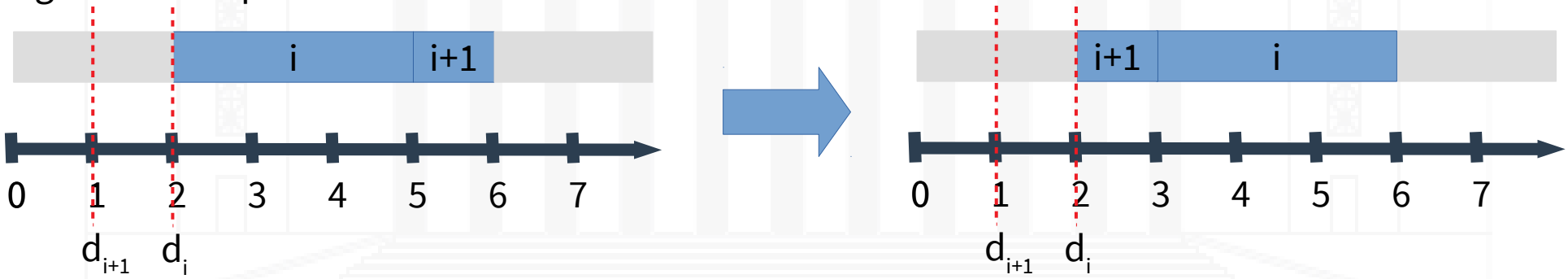
Casos de inversiones (cont.)



Ambos deadline después del termino de i+1
Intercambiar no empeora la latencia total.
Sigue siendo optimo

$L'_{i+1} = 2 + 1 - 6 < 0 \rightarrow l_{i+1}=0$

$L'_i = 3 + 3 - 7 < 0 \rightarrow l_i=0$



$L_{i+1} = 5 + 1 - 1 = 5$

$L_i = 2 + 3 - 2 = 3$

Mejoran ambos.
Podría empeorar?

$L'_{i+1} = 2 + 1 - 1 = 2$

$L'_i = 3 + 3 - 2 = 4$

Inversión - Generalización

Partimos de

O óptimo con inversiones

Llamamos

O' al optimo luego de intercambio

Podemos ver que

La tarea i+1 terminara antes. Su latencia se mantiene o disminuye.

La tarea i puede aumentar su latencia.

Puede aumentar el máximo de la programación?

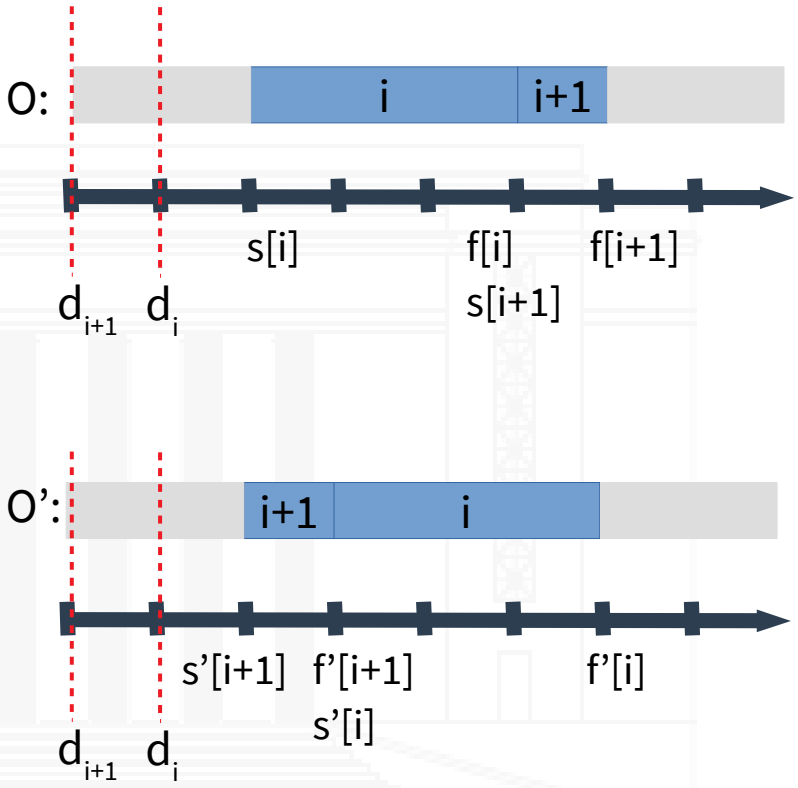
l'[i]=f'[i]-d[i]

Vemos que f[i+1] = f'[i]

Por lo tanto l'[i]=f[i+1]-d[i]

Como d[i]>d[i+1]

Entonces l'[i]=f[i+1]-d[i] < f[i+1]-d[i+1] = l[i+1] → **l'[i] < l[i+1]**



Una resolver una inversión mejora o mantiene la latencia máxima

Programación optima sin inversiones

Como

$L \geq l_{i+1} > l'_i$

Entonces

El intercambio no incrementa la latencia máxima de la programación

Mientras exista una inversión

Realizaremos otros intercambio.

Al finalizar

nos quedaremos con una programación optima, sin inversiones. (*2)

Colorario

Como

Existe un optimo sin tiempos muertos y sin inversiones (*2)

Y

Cualquier programación sin tiempos muertos e inversiones es equivalente o otra programación sin tiempos muertos e inversiones (*1)

Y

El algoritmo greedy retorna una programación sin tiempos muertos e inversiones

Entonces

La solución entregada por greedy es óptima

Complejidad

Complejidad temporal

Ordenar los pedidos: O(n log n)

Iterar sobre ellos O(n)

Complejidad total: O(n log n)

Complejidad espacial:

Almacenar los intervalos o(n)

```
Ordenar los pedidos en función de su deadline
(Asumiremos por simplicidad la notación d_1 ≤
d_2 ≤ … ≤ d_n)

Inicialmente f=s    // s es la hora de inicio
(podemos suponerlo como 0)
Desde el pedido i=1 al n
     Asignar el pedido i al intervalo s(i)=f a
f(i)=f+t_i
     f = f + t_i

Retornar el set de intervalos programados
[s(i), f(i)] para i=1 a n
```

.UBAfiuba

FACULTAD DE INGENIERÍA

Presentación realizada en Abril de 2021

.UBAfiuba
FACULTAD DE INGENIERÍA

Greedy: Mochila fraccionaria y cambio en monedas

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Algoritmos greedy

Paradigma de resolución de problemas

Utiliza una heurística de selección

En cada paso elige la mejor solución local (optimo local)

El criterio de selección depende únicamente de elecciones anteriores y/o de estado actual.

Con el objetivo de encontrar el optimo global

En general se puede crear para cualquier problema un algoritmo greedy

No todos los problemas admite soluciona optimas utilizándolos

Subestructura óptima

Un problema tiene subestructura optima

Si una solución optima al problema contiene dentro soluciones optimas a sus subproblemas

La solución greedy

Toma decisiones heurísticas para ir solucionando subproblemas de forma óptima para llegar a la solución óptima general

Mochila fraccionaria

Contamos con

un contenedor (mochila) con capacidad W

conjunto de n elementos fraccionables de valor v_i y peso w_i

Queremos

Seleccionar un subconjunto de elementos o fracciones de ellos de modo de maximizar el valor almacenado y sin superar la capacidad de la mochila

Solución Greedy

Priorizar los elementos mas valiosos por unidad

Llenar el contenedor con la mayor cantidad posible de unidades del elemento disponible mas valioso por unidad

Repetir el proceso mientras quede espacio en el contenedor y elementos disponibles

Complejidad

Ordenar los elementos según valor por unidad

O(nlogn)

Iterar mientras haya lugar

O(n)

La complejidad algorítmica de la solución es O(nlogn)

```
Lugar = W
Valor = 0
Mientras existan elementos disponibles y
                                    lugar > 0

    Sea x el elemento disponible con mayor
                                valor por unidad

    Cantidad = min(Wx , Lugar)

    Valor += Cantidad * Vx
    Lugar -= Cantidad
    Quitar x de disponible

Retornar Valor
```

Optimalidad

El algoritmos greedy es siempre optimo?

Llamaremos a la solución greedy G, y a los elementos elegidos $g_1, g_2, \ldots g_r$

Imaginemos que existe una solución optima O, con elementos $o_1, o_2, \ldots, o_s$

Si valor(O) > valor(G) entonces existe al menos un elemento o_i

que no esta en G con mayor valor por unidad que algún elemento gi en G

O esta con menor cantidad en G y esa diferencia esta en al menos un elemento en O en menor valor

Esto implica que Greedy no seleccionó un elemento disponible según su programación

Es absurdo! Por lo tanto greedy es optimo

Cambio mínimo

Contamos con

Un conjunto de monedas de diferente denominación sin restricción de cantidad

$\$ = (c_1, c_2, c_3, \ldots, C_n)$

Un importe X de cambio a dar

Queremos

Entregar la menor cantidad posible de monedas como cambio

Ejemplos

Denominaciones de las monedas

(1,5,10,25,50,100)

(!) Asumiremos que existe siempre la moneda unidad

Cambio mínimo a retornar

80 → (0,1,0,1,1,0)

24 → (4,0,2,0,0,0)

101 → (1,0,0,0,0,1)

Solución Greedy

Conocida como “solución del cajero”

Tomar la mayor denominación posible menor al cambio a dar

Dar tantas monedas de esa denominación como sea posible

Con el resto, repetir el procedimiento hasta dar todo el cambio

```
Cambio = X
#Monedas = 0
Mientras X > 0

     Sea Ci la denominación con mayor valor
                    en $ tal que Ci<=Cambio

          Cantidad = piso(Cambio / Ci)
          Cambio -= Ci * cantidad
          #Monedas += Cantidad


Retornar #Monedas
```

Complejidad O(n)

Optimalidad

Contra ejemplo

$ = (1,2,4,5,10)

X= 8

Greedy = (1,1,0,1,0) → 3

Optimo = (0,0,2,0,0) → 2

Greedy no es optimo!

Casos especiales

Existen casos donde el algoritmo greedy funciona

La base (1,5,10,25,50,100) es un caso

También funciona $=(1,5,25), pero no $=(1,10,25)

Se conoce a un sistema $ como “canonico”, “standard”, “ordenado” o “greedy”

a aquel en el que para todo x, greedy($,x) = optimo($,x)

(Existen diferentes papers que usan equivalentemente estos nombres)

Basta con un contraejemplo para determinar que un sistema no lo es

Cuando funciona greedy?

No es necesario probar todos los valores posibles de x

En “Optimal Bounds for the ChangeMaking Problem”

https://www.cs.cornell.edu/~kozen/Papers/change.pdf

Si existe un contraejemplo el mismo estará entre $c_3 + 1 < x < c_n + c_{n-1}$

En “A Polynomial-time Algorithm for the Change-Making Problem”

https://ecommons.cornell.edu/handle/1813/6219

Se presenta un algoritmo $O(n^3)$ para determinar si un $ es canónico

Se puede construir un algoritmo siempre optimo para resolver este problema utilizando “programación dinámica”

.UBAfiuba

FACULTAD DE INGENIERÍA

Presentación realizada en Abril de 2020

.UBAfiuba
FACULTAD DE INGENIERÍA

Greedy: Seam carving y camino mínimo

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Adecuación inteligente de imágenes

Los contenidos web modernos

están preparados para verse en diferentes dispositivos.

Cada dispositivo tiene tamaños y escalas de pantallas diferentes

el contenido “responsive” se acomoda a la relación de aspecto y dimensiones

Una misma imagen debe verse bien en cada situación

Se realiza redimensión y recorte para ajustar al contenedor

A veces esto no arroja resultados satisfactorios.

Resultados insatisfactorios

Imagen original

Escalado

Recorte

Seam Carving

Algoritmo para la manipulación de la imagen

Creado por Shai Avidan y Ariel Shamir en 2007

Paper “Seam Carving for content-aware image Resizing”

Enlace: *https://dl.acm.org/doi/pdf/10.1145/1275808.1276390*

Analiza la imagen recortando los pixels de menor importancia

Encuentra una secuencia de pixel horizontal o vertical

Al retirarla tiene el menor impacto sobre la visualizacion

Seam: veta / Carving: tallado

Retira tantas vetas como sea necesario para llegar al tamaño requerido

Ejemplo

Imagen original

Escalado

Recorte

Seam carving

Pixels poco importantes

Existen diferentes métodos

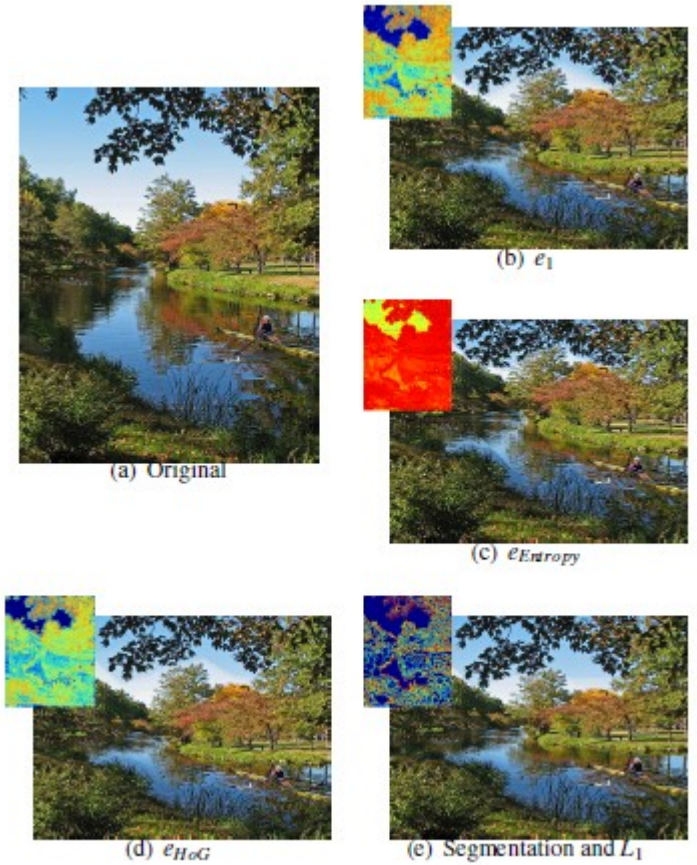
Por ejemplo: entropía, diferencia cromática, movimiento de la mirada, etc

Diferencia cromática

Pixel rodeados de pixels similares → poca importancia (ej: cielo azul)

Pixel rodeado de pixels diferentes → muchas importancia (ej: cara de una persona)

Inicialmente debo calcular la importancia de cada pixel



Vetas

Pueden ser

- Horizontales
- Verticales

Buscaremos una veta

- Que inicie en extremo superior (izquierdo)
- Finalice en el extremo inferior (derecho)
- Cuya suma de importancia sea la mejor posible

Camino mínimo

Trataremos la imagen

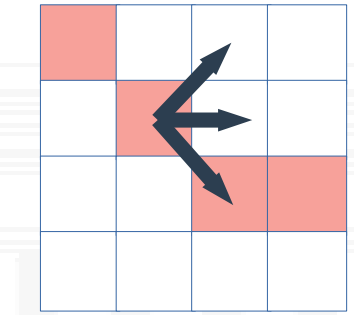
Como una grilla de pixels inter comunicados

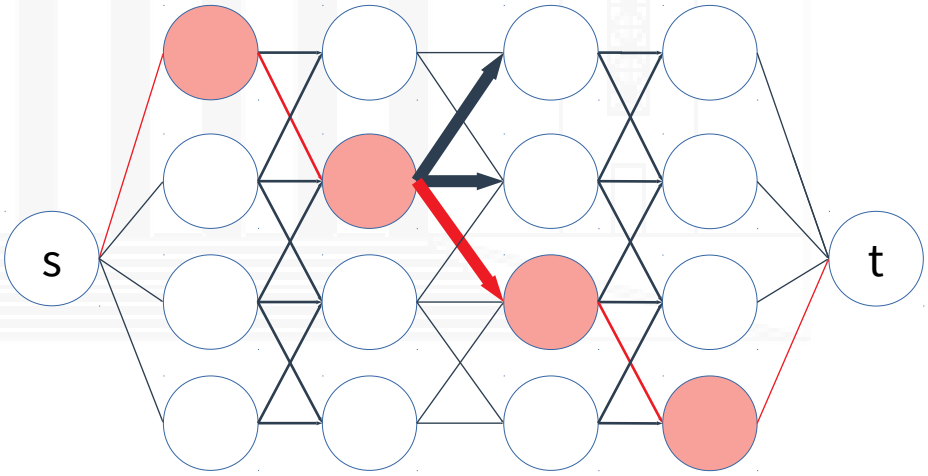
Lo representaremos como un grafo

Los pixels son nodos

Los ejes son los posibles caminos de la veta

Calcularemos el camino mínimo entre s y t





Dijkstra

Propuesto por Edsger Dijkstra en 1959

Es Greedy

Funciona para un grafo G

Dirigido y ponderado (con costos positivos)

de N nodos no aislados (en nuestro caso es la cantidad de pixels) y M ejes

Dados dos nodos

“s” inicial

“t” final

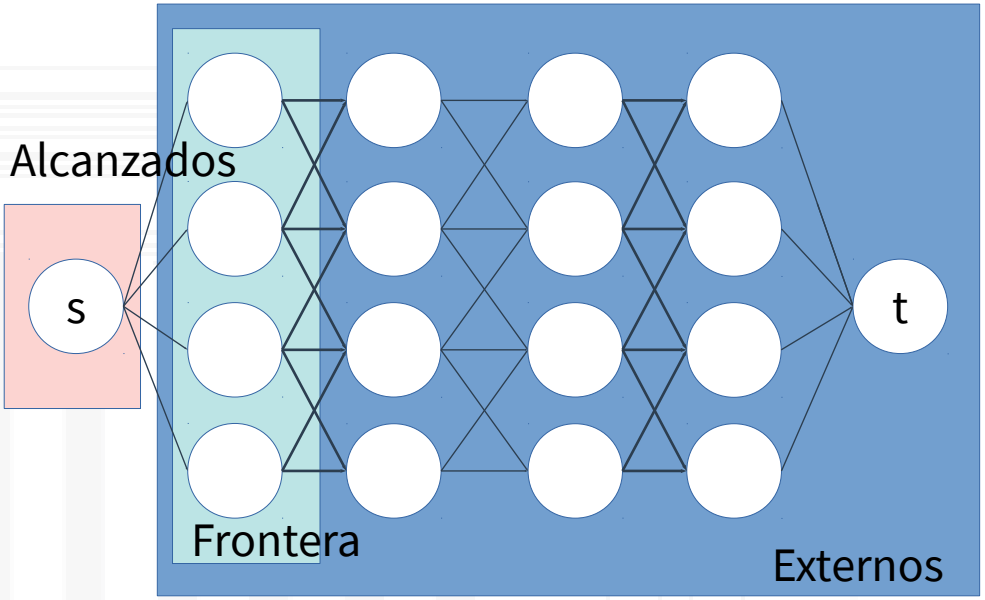
Encuentra el camino mínimo que los une

Funcionamiento

Es un algoritmo iterativo

Divide los nodos en 3 conjuntos

- “alcanzados”: inicialmente solo el nodo “s”
- “externos”: inicialmente todos los nodos menos “s”.
- “frontera”: pertenecientes a externos que están “conectados” a algún nodo de los "alcanzados”



Funcionamiento (cont.)

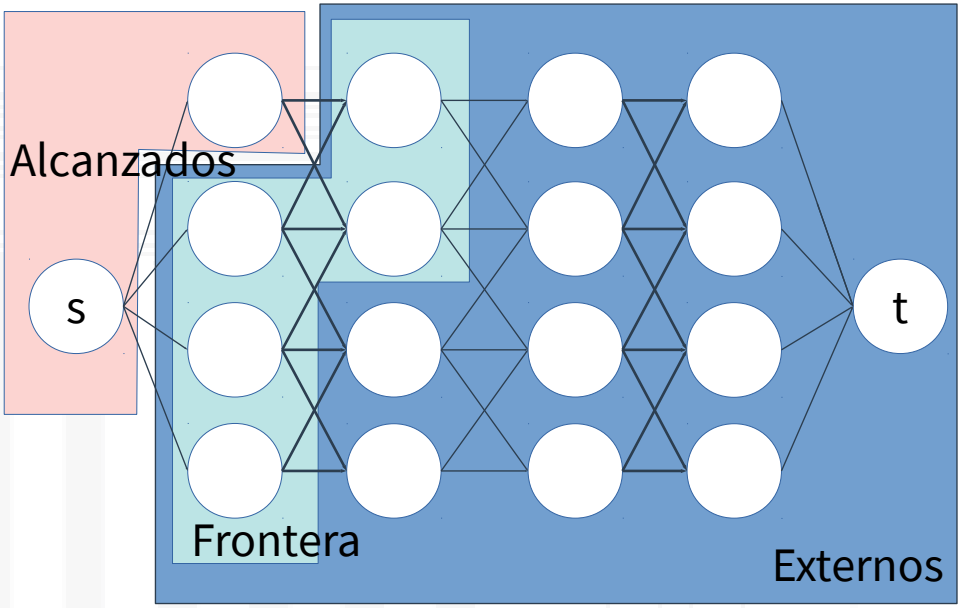
En cada iteración

Se obtiene aquel nodo x de la frontera cuyo costo de llegada desde “s” sea el menor posible (elección greedy)

Se agrega x con su costo al conjunto de “alcanzados” (se registra desde que nodo se llegó)

Se pasan los nodos”externos” con conexión a x a la “frontera” con su cálculo de costo de llegada

Se actualizan los costos de los nodos de la frontera con conexión a x si el nuevo costo es menor.



Funcionamiento (cont.)

Finalización

El algoritmo finaliza cuando no quedan nodos en la frontera

(o cuando se llega al nodo “t”)

Costos de llegada al nodo

Cuando un nodo se encuentra en los alcanzados su costo no puede modificarse y es el mínimo.

Cuando se encuentra en la frontera, es el mínimo costo entre todos los costos de sus nodos vecinos alcanzados mas el costo de llegar a él desde ellos

Cuando es externo su costo es infinito

Pseudocódigo

```
Alcanzados = {(s,0,_)}
frontera = ∅
Por cada nodo x vecino a S
    Agregar a frontera x con costo Cx=Csx y predecesor(x)=s

Mientras la frontera ≠ ∅
    Sea x nodo en frontera con menor costo
    Quitar x de frontera
    Alcanzados = Alcanzados ∪ {(x,Cx,predecesor(x))}
    Por cada y vecino de x
        cy' = Cx + Cxy
        Si y ∈ frontera y cy' < cy
            predecesor(y) = x
            actualizar en frontera y con costo cy'
        Si y ∉ frontera
            predecesor(y) = x
            agregar en frontera y con costo cy'
```

Implementación

Frontera

Heap de mínimos con actualización de clave

Alcanzados

Vector de tamaño “n”

Vecinos de cada nodo

Lista de adyacencias por nodo.

```
Alcanzados = {(s,0,_)}
frontera = ∅
Por cada nodo x vecino a S
     Agregar a frontera x con costo Cx=Csx
                              y predecesor(x)=s

Mientras la frontera ≠ ∅
     Sea x nodo en frontera con menor
                                    costo
     Quitar x de frontera
     Alcanzados = Alcanzados ∪
                  {(x,Cx,predecesor(x))}
     Por cada y vecino de x
          cy' = Cx + Cxy
          Si y ∈ frontera y cy' < cy
               predecesor(y) = x
               actualizar en frontera y con
                                        costo cy'
          Si y ∉ frontera
               predecesor(y) = x
               agregar en frontera y con
                                    costo cy'
```

Complejidad algorítmica

El loop se ejecuta n-1 veces

Cada vez obtengo el nodo con menor en O(logn) ← extract_min

En la frontera se inserta n-1 veces

con costo O(logn) ← insert

En la frontera se actualiza (en el peor de los casos) m veces

con costo O(logn) ← dec_key

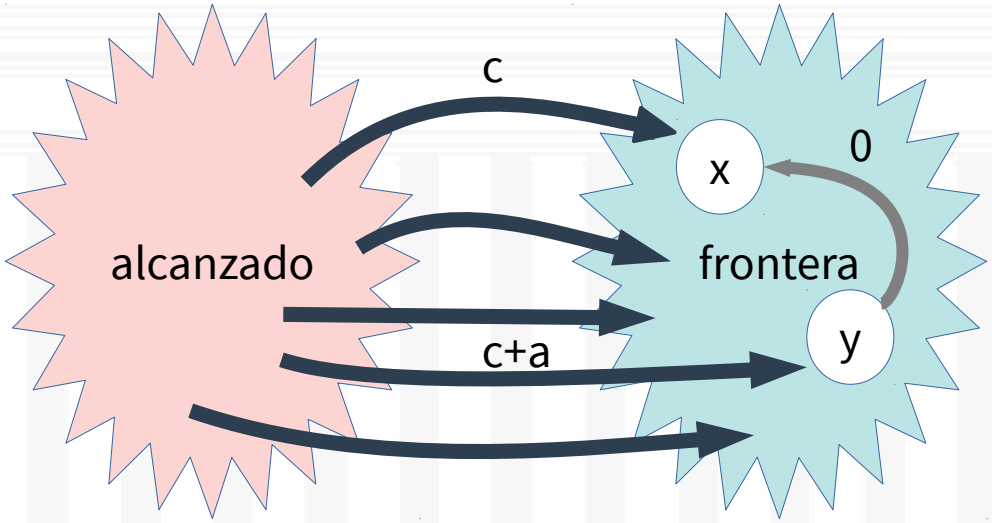
La complejidad es O([n+m]logn)

```
Alcanzados = {(s,0,_)}
frontera = ∅
Por cada nodo x vecino a S
     Agregar a frontera x con costo Cx=Csx
                              y predecesor(x)=s

Mientras la frontera ≠ ∅
     Sea x nodo en frontera con menor
                                      costo

     Quitar x de frontera
     Alcanzados = Alcanzados ∪
                  {(x,Cx,predecesor(x))}
     Por cada y vecino de x
          cy' = Cx + Cxy
          Si y ∈ frontera y cy' < cy
               predecesor(y) = x
               actualizar en frontera y con
                                      costo cy'
          Si y ∉ frontera
               predecesor(y) = x
               agregar en frontera y con
                                  costo cy'
```

Elección Greedy

En cada iteración selecciona el menor costo disponible

Puede esto resultar contraproducente?



En el “mejor de los casos” puedo llegar a nodo x otro nodo en la frontera con costo 0

El costo acumulado de y es superior al camino directo a x en “a”

La elección greedy es correcta!

.UBAfiuba

FACULTAD DE INGENIERÍA

Presentación realizada en Abril de 2020

.UBAfiuba
FACULTAD DE INGENIERÍA

Árbol recubridor mínimo

Teoría de Algoritmos I (75.29 / 95.06)

Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Problema

Sea

G=(V,E) un grafo conexo y ponderado (con costos “w” positivos)

Queremos

Seleccionar un subconjunto de ejes $T \subseteq E$

De forma tal

Que el nuevo grafo G’=(V,T) sea conexo

Y el costo total $W = \sum_{e \in T} w_e$

sea mínimo entre todos los posibles grafos conexos

El resultado es un árbol

Un árbol

es un grafo simple no dirigido conexo y sin ciclos

El grafo G'=(V,T)

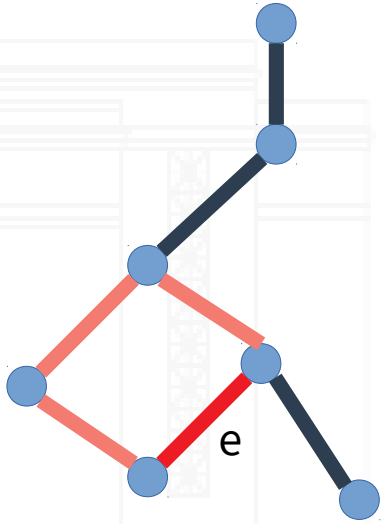
Por definición es conexo

Si G' tiene un ciclo C

Sea un eje e ∈ C,

Si extraemos e del grafo G', el grafo resultante aun sera conexo

Y tendrá un costo menor.



Por lo tango G' no tiene ciclos

Árbol recubridor mínimo

Se conoce el problema con el nombre

Árbol recubridor mínimo

Para un grafo G=(V,E)

existen varios posibles arboles rucubridores

Solo

un subconjunto de ellos (o 1) es mínimo

Propiedad de corte (cut property)

Supongamos por un instante

Que todos los costos de los ejes son diferentes

Para cualquier

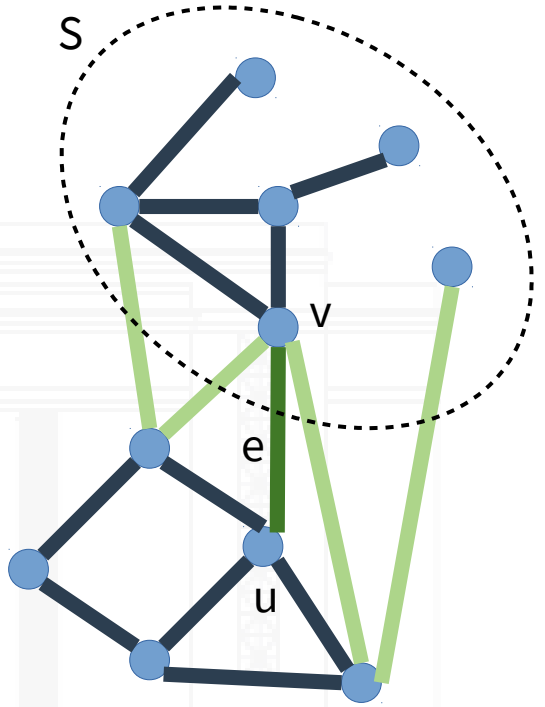
subset de nodos $S \subset V$ (con $S \neq V$ y $S \neq \emptyset$)

Existe un subconjunto de ejes F

Tal que para todo $f=(u,v) \in F$, $u \in S$ y $v \in V-S$

Existe un eje e = (u,v) ∈ F

Que tiene el costo menor que el resto de los demás en F



Propiedad de corte (cont.)

Sea un árbol recubridor T que no contiene a e ∈ F entre sus ejes

Pero que contiene e'=(u',v') ∈ F siendo "puente" entre S y V-S

Si intercambiamos e' por e

conformando un nuevo grafo sin ciclos y conexo (un árbol) *

Donde ahora e será "puente" entre S y V-S

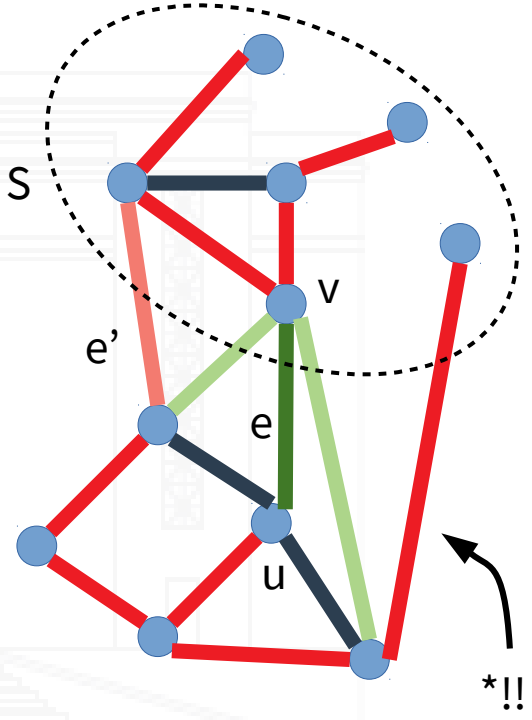
Sabemos que $W_e < W_{e'}$

Por lo que el costo del árbol recubridor resultante T' será menor.

En conclusión

Cualquier árbol recubridor mínimo de G deberá tener a e como puente entre S y V-S

(para cualquier S que tenga a e en la frontera)



Algoritmos tipo greedy

Para encontrar un árbol recubridor mínimo

- Existen varios algoritmos greedy (todos ellos óptimos)

Algunos de ellos

- Algoritmo Kruskal
- Algoritmo Prim
- Algoritmo de eliminación inversa (Reverse-delete)
- ...

Todos ellos son iterativos

- Funcionan agregando/quitando un eje a la vez
- Basándose en la propiedad de corte

Algoritmo Kruskal

Algoritmo propuesto por Joseph Kruskal en 1956

En la publicación “On the shortest spanning subtree and the traveling salesman problem”

Inicialmente

Todos los nodos de G=(V,E) conforman arboles en un bosque

Iterativamente en orden creciente de costo

recorre los ejes de E

En una iteración, analizando el eje e=(u,v)

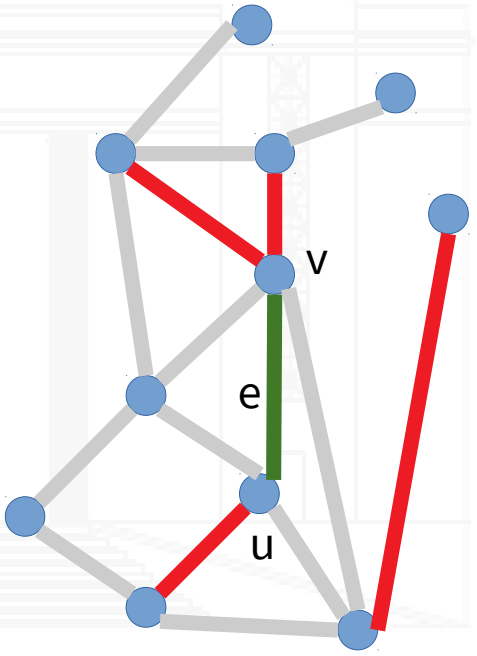
Evalúa unir los arboles que contienen a u y v en uno solo.

Desecha la acción Si el resultado de esa operación

se realiza en nodos u y v que ya pertenecen al mismo árbol

(agregarlo generaría un ciclo)

El resultado es un árbol recubridor mínimo



Kruskal - Optimalidad

Queremos ver que

Cada vez que el algoritmo agrega un eje, lo hace de a cuerdo a la propiedad de corte

Sea

e=(v,w) eje agregado en una iteración

S el subconjunto de nodos a los cuales v tiene un camino antes de agregar el eje e

Sabemos que

$v \in V$ y que $w \notin V$ (sino se crearía un ciclo al agregar el eje e)

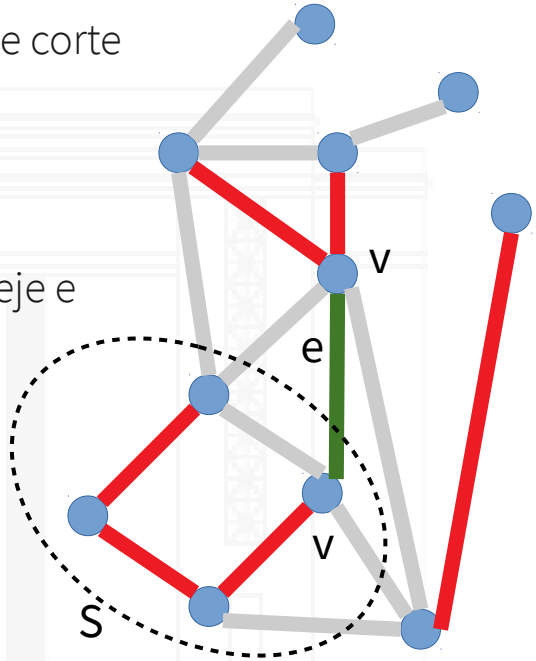
No hay ejes seleccionados que vayan de S a V-S

Por lo tanto

Por como se selecciona e, corresponde al eje menos costoso que une S con V-S

En conclusión

El eje e pertenece a cualquier árbol recubridor mínimo de G (según propiedad corte)



Algoritmo Prim

Algoritmo fue diseñado en forma independiente por

Vojtech Jarnik (1930), Robert C. Prim (1957) y Dijkstra (1959)

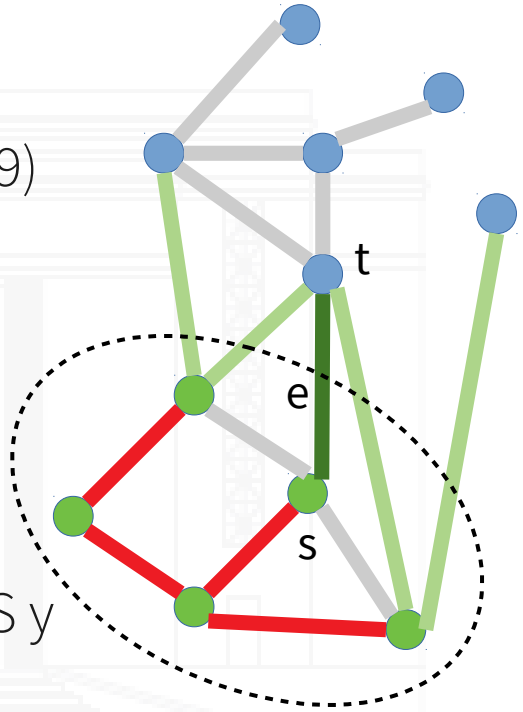
Inicialmente

Se selecciona un u nodo del grafo y S={u}

Repetimos mientras V-S ≠ ∅

Seleccionamos el eje e=(s,t) con el menor costo y con s ∈ S y t ∈ V-S

Agregamos t a S (S=S+{t})



Prim - Optimalidad

Queremos ver que

Cada vez que el algoritmo agrega un eje, lo hace de a cuerdo a la propiedad de corte

Sea

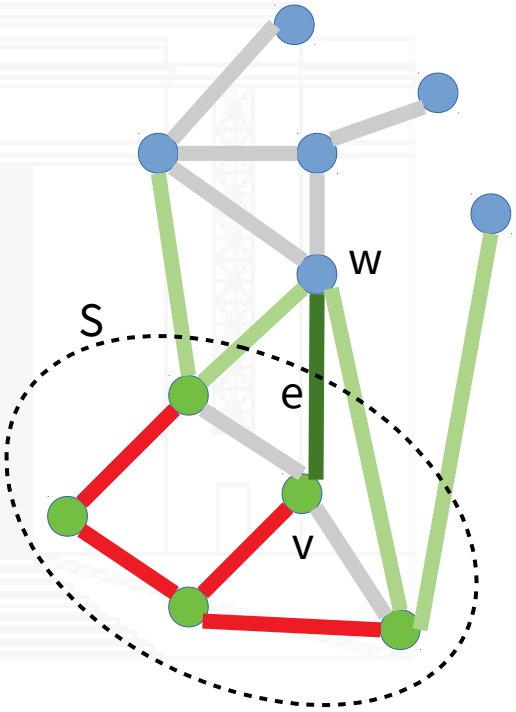
e=(v,w) eje agregado en una iteración

S por funcionamiento del algoritmo es un árbol y $v \in S$

V-S son todos nodos sueltos que aun no se unen al arbol S

Según algoritmo $w \in V-S$

El eje e es aquel de menor costo que va de V a V-S



En conclusión

El eje e pertenece a cualquier árbol recubridor mínimo de G (según propiedad corte)

Algoritmo de eliminación inversa

Inicialmente

Comenzamos con el grafo completo

Iterativamente en orden decreciente de costo

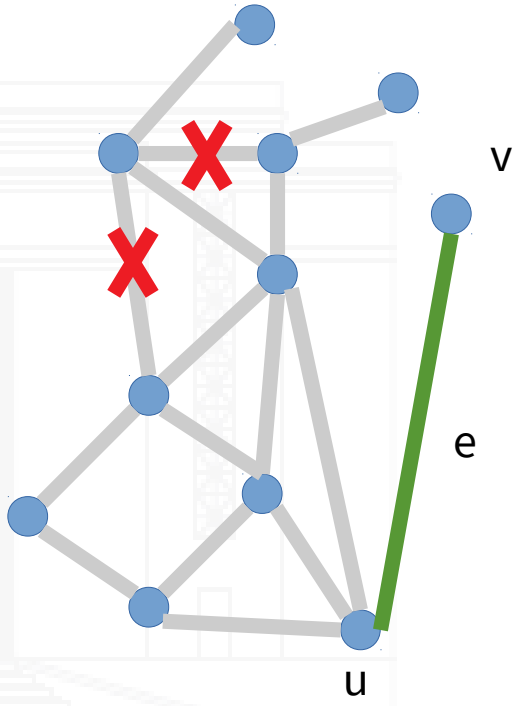
recorre los ejes de E

En una iteración, analizando el eje e=(u,v)

Eliminamos el eje si al realizarlo el grafo resultante continua siendo conexo

Sino lo mantenemos

El resultado es un árbol recubridor mínimo



Propiedad de ciclo

Supongamos por un instante

Que todos los costos de los ejes son diferentes

Sea

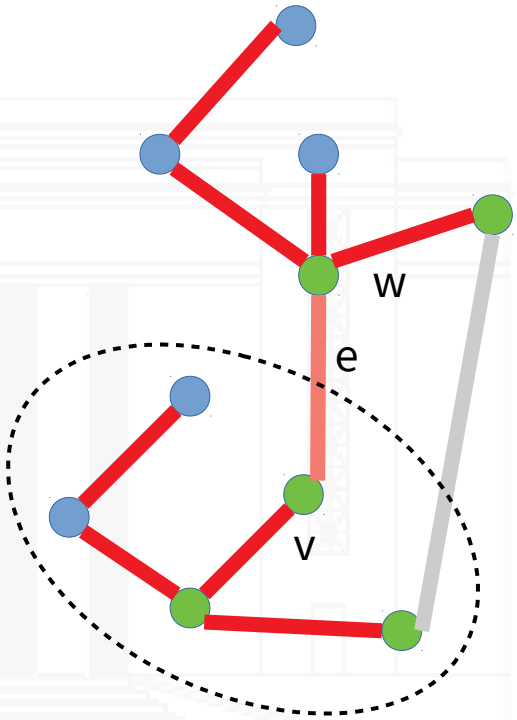
C un ciclo en G

e=(v,w) el eje mas costoso dentro de C

T un árbol recubridor que contiene a e

Si eliminamos e de T

Nos quedarán 2 componentes conexos S y V-S



Propiedad de ciclo (cont)

Para volver a generar el árbol

debemos seleccionar un eje e'=(v',w') con v' ∈ S y w' ∈ V-S

Como e pertenece al ciclo C,

existe en el gráfo otro eje con menor costo que construye un camino entre v y w

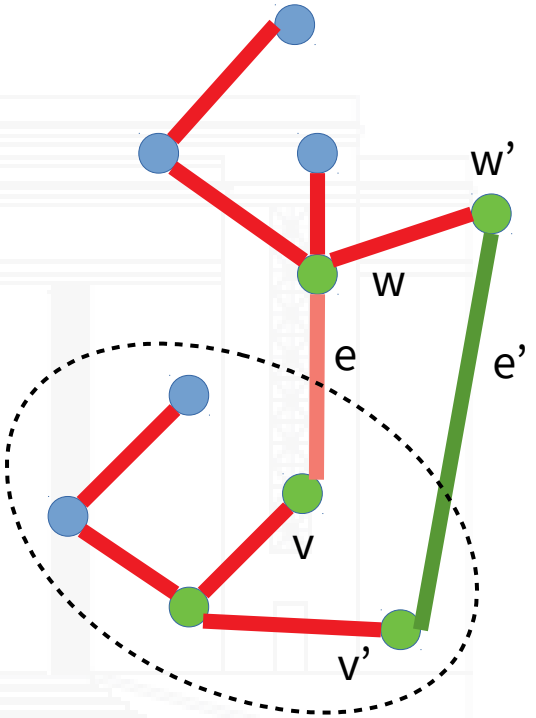
Ese eje debe ser e'!

Al agregar e'

Se genera un nuevo árbol recubridor T' de menor costo que T

Por lo tanto

Si existe un ciclo C en el Grafo G, el eje de mayor costo en C no pertenece al árbol recubridor mínimo



Eliminación inversa - Optimalidad

Sea

El eje e=(v,w) eliminado por el algoritmo en una iteración

Como

Es el eje de mayor costo que aun no se ha removido (obviando a aquellos que quitarlo generarán una desconexión del grafo)

Y al quitar el eje el algoritmo se asegura que el resultante siga siendo conexo

Entonces

El eje e pertenece a un ciclo inmediatamente antes de su remoción.

Por lo tanto

El algoritmo elimina aquellos ejes que no pertenecen al árbol recubridor mínimo (por propiedad de ciclo)

¿Qué pasa si tengo ejes con costos iguales?

Si

Varios ejes pueden compartir el mismo costo

Seleccionar

Cualquiera de ellos nos lleva al árbol recubridor mínimo

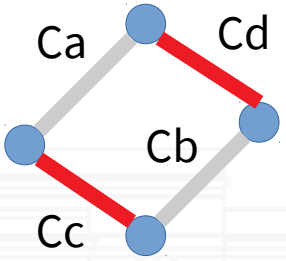
De hecho

Esto hace que puedan existir varios árboles recubridores mínimos

Necesitamos

poder desempatar entre entre los ejes de igual valor para ver cual quitar

(puede ser simplemente por orden en el que vienen)



Cc < Cd < Ca = Cb

.UBAfiuba

FACULTAD DE INGENIERÍA

Presentación realizada en Septiembre de 2020

.UBAfiuba
FACULTAD DE INGENIERÍA

Greedy: Códigos de Huffman

Teoría de Algoritmos I (75.29 / 95.06)
Ing. Víctor Daniel Podberezski
vpodberezski@fi.uba.ar

Compresión de datos

Reducción del volumen de datos

para representar una determinada información empleando una menor cantidad de espacio

Llamaremos fuente a todo aquello que emita mensajes

Existe un conjunto finito de mensajes posibles

Los mensajes se representaran mediante códigos

El objetivo es elegir códigos de tal forma de reducir al mínimo el tamaño de lo emitido por la fuente

Códigos

Convención que establece como representar cada mensaje utilizando una combinación de símbolos.

Ejemplos: ASCII, Morse, ...

Pueden tener longitud fija o variable

No deben ser ambiguos!

Códigos decodificables

Para cualquier sucesión de códigos, solo existe un único conjunto de mensajes validos.

Los códigos de longitud fija son siempre decodificables

Código 1: ”010” puede ser: “AD” o “B”
Código 2: es decodificable (probar todas las combinaciones!)

Mensaje	Código 1	Código 2
A	0	10
B	010	00
C	01	11
D	10	110

Códigos prefijos

Son siempre decodificables

Un código es prefijo si no existe ningún código que tenga un prefijo igual a otro código completo

Mensaje	Código 2	Código 3
A	10	0
B	00	10
C	11	110
D	110	111

Códigos de Huffman

Presentado en 1952 por David Huffman

En paper “A method for the construction of minimum redundancy codes”

http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf

Son códigos de longitud variable y prefijos

La longitud de cada código está basada en su frecuencia de cada mensaje en el emisor (fuente)

Se basa en la teoría de información de Claude Shannon

Son óptimos: no existen otros códigos prefijos para la misma fuente que la codifique en menor longitud

Códigos de Huffman (cont.)

Sea Alfabeto A = (a_1, a_2, …, a_n)

Llamaremos W=(w_1,w_2,…, w_n) al peso (frecuencia) de cada a_i

Construiremos C(W) = (c_1,c_2,..c_n) códigos prefijos y binarios

Llamamos:

$$\text{Longitud}\left(C\left(W\right)\right)=\sum_{i=1}^{n} w_i * \text{size}\left(C_i\right)$$

De tal forma que:

$$\text{Longitud}\left(C\left(W\right)\right) \leqslant \text{Longitud}\left(T\left(W\right)\right) \text{ para cualquier otro código prefijo T(W)}$$

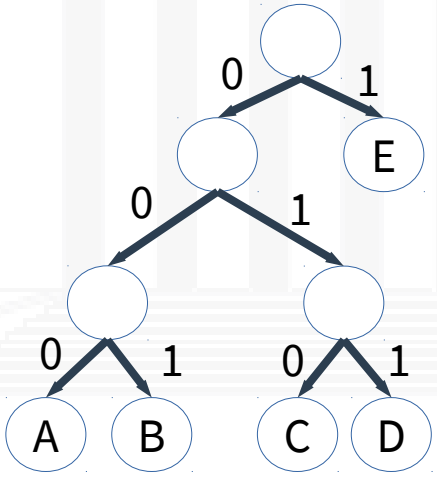
Código prefijo: Árbol binario

Podemos representar un código prefijo mediante un árbol binario

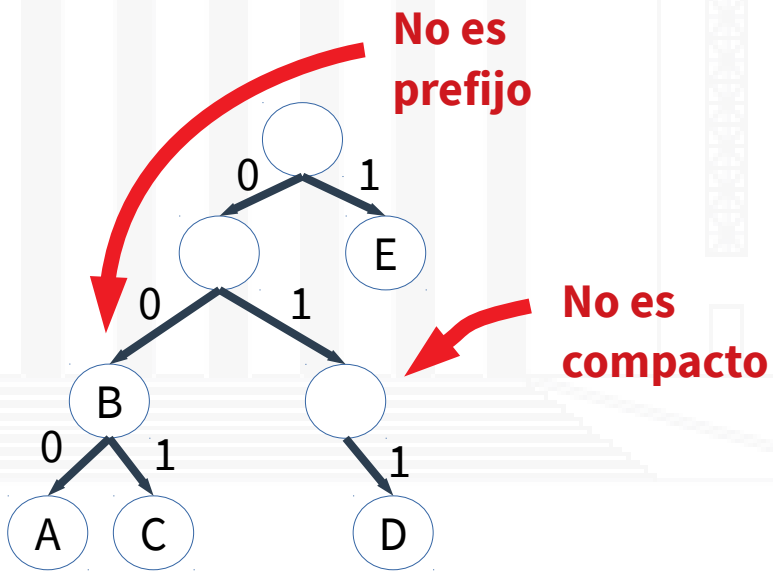
Las hojas son los códigos

Cada nodo tiene 2 o ningún descendiente

A=000
B=001
C=010
D=011
E=1



Válido



Inválido

Algoritmo Greedy

Genera un árbol de “huffman”

Para armar el optimal prefix code

Inicialmente

cada código ci es un nodo hoja con peso wi

Mientras quede más de un nodo sin padre

Toma los dos nodos x, y de menor peso sin padre.

Crear un nuevo nodo z con wz = wx + wy

Definir a z como padre de x e y

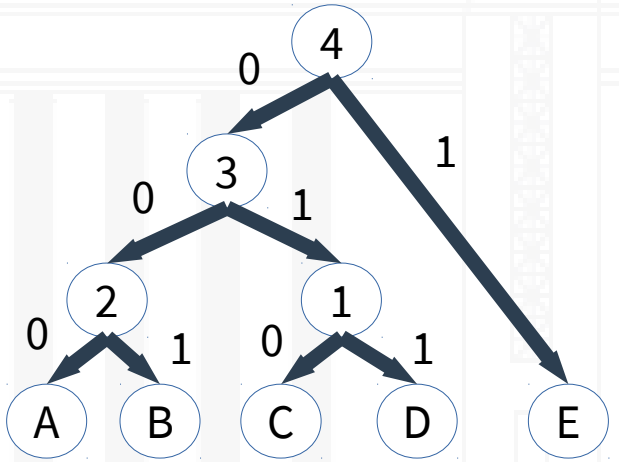
El ultimo nodo sin padre sera la raíz del árbol

Fuente: ABEEECAEEEDBEEEE

A → wa = 2 B → wb = 2

C → wc = 1 D → wd = 1

E--→ we = 10



w1 = wc + wd = 2 w3 = w1 + w2 = 6

w2 = wa + wb = 4 w4 = w3 + we = 16

A=000 C=010

B=001 D=011

E=1

Algoritmo Greedy

$$\text{Longitud}(C(W)) = \sum_{i=1}^{n} w_i * \text{size}(C_i)$$

i	Cod	w	size
A	000	2	3
B	001	2	3
C	010	1	3
D	011	1	3
E	1	10	1

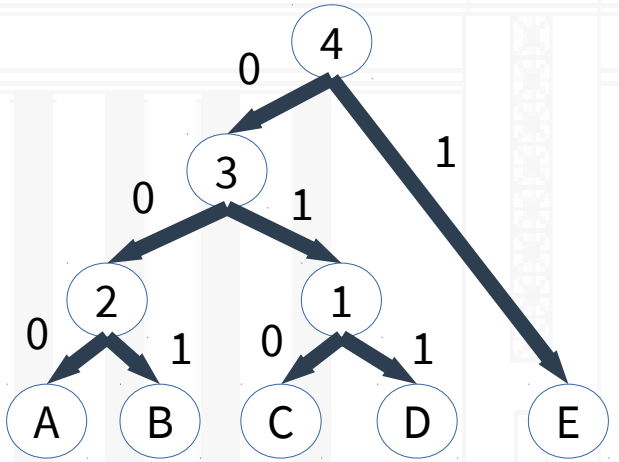
2*3 + 2*3 + 1*3 + 1*3 + 10*1 = 28

Fuente: ABEEECAEEEDBEEEE

A → wa = 2 B → wb = 2

C → wc = 1 D → wd = 1

E--→ we = 10



w1 = wc + wd = 2 w3 = w1 + w2 = 6

w2 = wa + wb = 4 w4 = w3 + we = 16

A=000 C=010

B=001 D=011

E=1

Implementación

Utilizaremos un Heap de mínimos

- El nodo del árbol será el elemento
- La frecuencia será la clave

En cada iteración

- Se obtienen los 2 nodos de menor peso
- Se ingresará un nuevo creado

El ultimo elemento en le heap

- es la raíz del árbol

La complejidad algorítmica es

- O(nlogn)

```
Desde i=1 a n
     Crear nodo z
     z.char = a[i]
     z.w = w[i]

     Heap.add(z,z.w)

Desde i=1 a n-1
     x = Heap.get()
     y = Heap.get()

     Crear nodo z
     z.left = x
     z.right = y
     z.w = x.w + y.w

     Heap.add(z,z.w)

Retornar Heap.get()
```

Optimalidad

Para probar que nuestro resultado es optimo debemos realizar 2 demostraciones

Prueba de selección greedy

Lograr mostrar que elegir en los dos mensajes de menor peso nos acerca a la solución optima global

Prueba de los subproblemas

Lograr demostrar que el subproblema derivado de nuestra elección se puede solucionar mediante la misma selección greedy

Selección greedy

Sea

$A=(a_1,a_2,\ldots,a_n)$ alfabeto con pesos $W=(w_1,w_2,\ldots,w_n)$

Sean a y b $\in$ A los 2 mensajes con menor frecuencia de la colección

Existe un código prefijo óptimo tal que

$size(C_a) = Size(C_b)$ y solo difieren en su ultimo bit

Ademas $size(C_a) = size(C_a) \geq size(C_i)$ con $i \in A - \{a,b\}$

Selección greedy - demostración

Sea

$x,y \in A$ siblings en hojas de máxima profundidad en árbol T

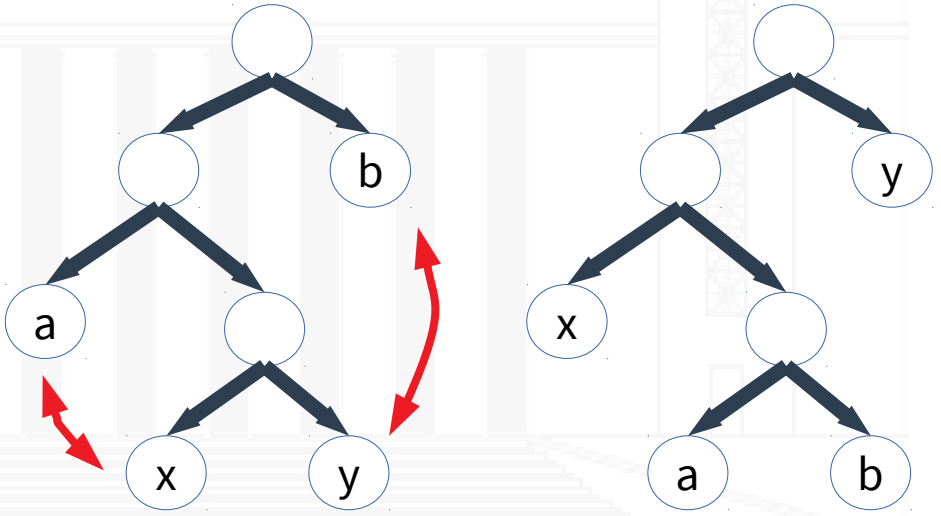
$A,b \in A$ tal que $w_x \geq w_y \geq w_b \geq w_a$

Intercambiamos

a con x

b con y

Llamaremos T' al nuevo árbol



Arbol T Arbol T'

Selección greedy – demostración (cont.)

Tenemos que

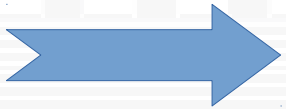
$$L(C(W)) = \sum_{i=1}^{n} w_i * \text{size}(C_i)$$

$$L(T(W)) - L(T'(W)) =$$

$$= w_a * \text{size}(a) + w_b * \text{size}(b) + w_x * \text{size}(x) + w_y * \text{size}(y)$$

$$- w_{a'} * \text{size}(a') + w_{b'} * \text{size}(b') + w_{x'} * \text{size}(x') + w_{y'} * \text{size}(y')$$

$$\text{size}(a) = \text{size}(x') \quad \text{size}(a') = \text{size}(x)$$
$$\text{size}(b) = \text{size}(y') \quad \text{size}(b') = \text{size}(y)$$



$$L(T(W)) - L(T'(W)) \geqslant 0$$

Disminuye o se mantiene el tamaño de la fuente comprimida

Prueba de los subproblemas

Sea

$A=(a_1,a_2,\ldots,a_n)$ alfabeto con pesos $W=(w_1,w_2,\ldots,w_n)$

Sean a y b $\in$ A los 2 mensajes con menor frecuencia de la colección

$A' = A - \{a,b\} + \{z\}$ tal que $w_z = w_a + w_b$ y el resto de los pesos iguales

Sea T' arbol representando un código prefijo óptimo para C'

Sea T arbol representando un código prefijo óptimo para C

Entonces

T se puede obtener de T' reemplazando el nodo hoja z por un nodo con hijos a y b

Prueba de los subproblemas - demostración

Por un lado podemos ver que

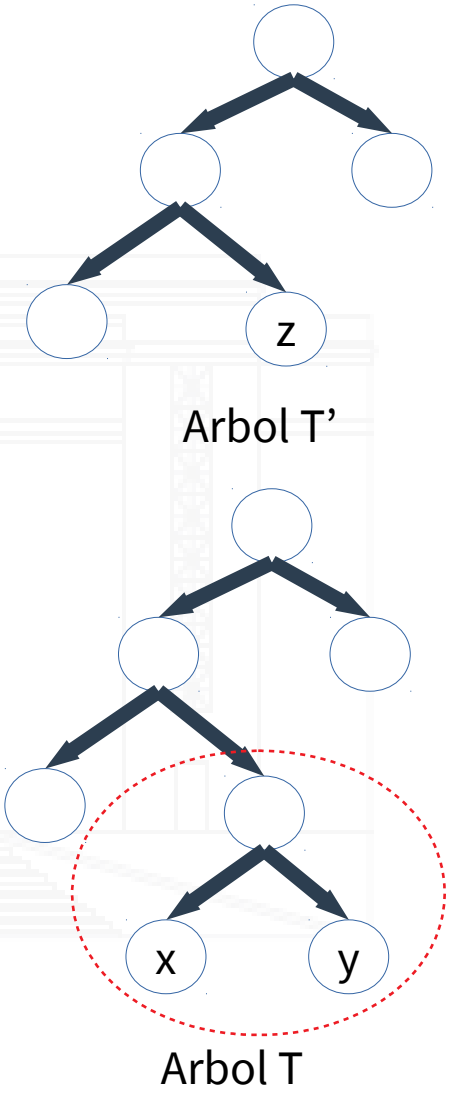
Todos los pesos y longitudes son iguales excepto a,b y z

$$\text{size}(a)=\text{size}(b)=\text{size}(z)+1$$

$$w_a*\text{size}(a)+w_b*\text{size}(b)=(w_a+w_b)*(\text{size}(z)+1)$$

$$=(w_z)*\text{size}(z)+(w_a+w_b)$$

$$L(T')=L(T)-(w_a+w_b)$$



Prueba de los subproblemas - demostración (cont.)

Supongamos que T no representa un código optimo prefijo de A

Entonces existe un T'' optimo tal que L(T'') < L(T)

T'' tiene que tener a y b como siblings (por demostración anterior)

Sea T''' el árbol T''con el padre de a y b reemplazado por la hoja z

Entonces

$L(T''') = L(T'') - wa - wb$

$< L(T) - wa - wb = L(T')$

Es una contradicción

Si T' es un código optimo, entonces debe tener la misma longitud de T'''

.UBAfiuba

FACULTAD DE INGENIERÍA

Presentación realizada en Abril de 2020