DECSAI
Departamento de Ciencias de la Computación e I.A.
Universidad de Granada

Scheduling

Fernando Berzal, berzal@acm.org

Scheduling

- Introducción
  - Uso de recursos
  - Notación
- Soluciones exactas
  - Algoritmos greedy
  - Programación dinámica
  - Branch & bound
- Técnicas heurísticas
  - Algoritmos greedy
  - Técnicas de búsqueda local

Scheduling

Definición

Scheduling = Planificación de acciones que requieren el uso de recursos.

Tipos de recursos

- Tiempo
- Maquinaria
- Personal
- Material
- Infraestructuras
- Energía
- Dinero

Scheduling

Solución a un problema de planificación:

- Plan (conjunto parcialmente ordenado de acciones)
- Acciones (operadores completamente instanciados) i.e. especificando el uso de recursos.

Con recursos…

- Podemos modelar los recursos como parámetros de las acciones…
  Demasiado ineficiente
  (el planificador intenta todas las combinaciones)

Scheduling

Enfoque alternativo: **planificar antes de asignar**

1. Planificación sin especificar el uso de recursos (variables asociadas a los recursos no instanciadas): Planificación → Razonamiento causal (qué hacer).

2. Asignación de recursos a las acciones: Scheduling → Asignación de recursos (y tiempos) i.e. cómo cuando hacerlo

No resulta óptimo ☹

Scheduling

Acciones y recursos

Recursos = Entidades necesarias para realizar acciones.

- Variables de estado
  (modificadas por las acciones en términos absolutos)
  p.ej. Mover recurso X de Lugar1 a Lugar2.

- Variables de uso de recursos
  (modificadas por las acciones en términos relativos)
  p.ej. Consumir X litros de gasolina hace que el nivel del depósito baje de L a L-X

Scheduling

Acciones con restricciones temporales

- Hora de comienzo más temprana: $start_{min}(a)$
- Hora de comienzo real: start(a)
- Hora límite de finalización: $end_{max}(a)$
- Hora de finalización real: end(a)
- Duración: duration(a)

Tipos de acciones

1. "Preemptive" (no pueden interrumpirse)
   duration(a) = end(a) - start(a)
2. "Non-preemptive"
   (pueden interrumpirse, lo que permite a otras acciones utilizar sus recursos mientras dure la interrupción)

Scheduling

Acciones que utilizan recursos

- Tipo de recurso requerido (r)
- Cantidad requerida del recurso (q)

Tipos de recursos

- Reutilizables (el recurso queda disponible para otras acciones cuando se completa la acción).

  p.ej. herramientas, máquinas, espacio de almacenamiento…

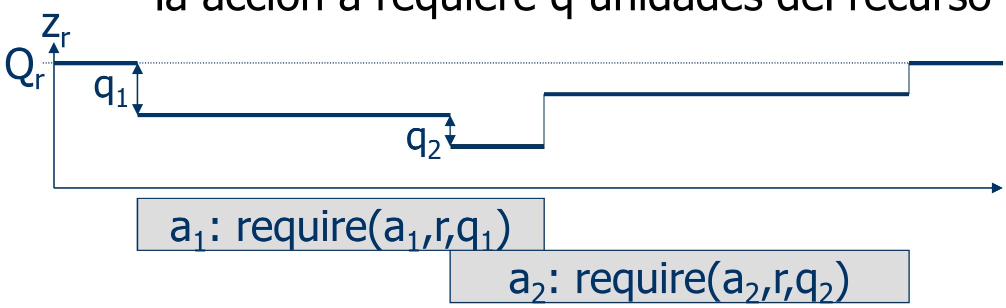
- Consumibles
  (el recurso se gasta cuando se termina la acción).

  p.ej. energía, tiempo de CPU, crédito…

Scheduling

Recursos reutilizables

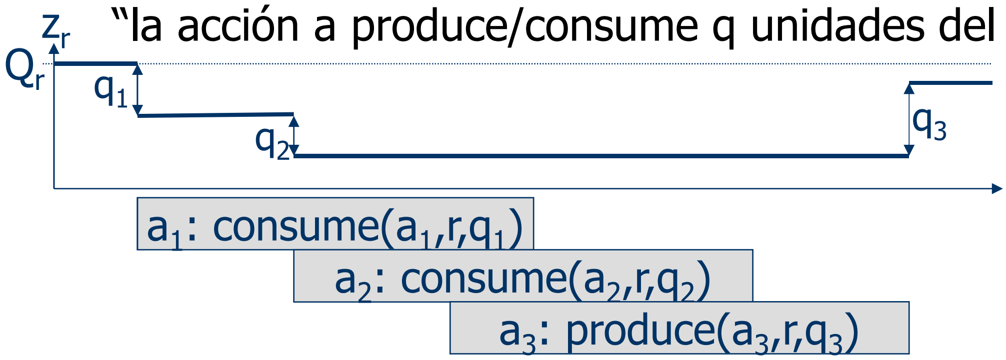
- Disponibilidad de recursos
  - Q_r Cantidad total del recurso r.
  - $z_r(t)$ Nivel del recurso r en el instante t.
- Requisitos de uso de recursos:
  - require(a,r,q)
  - “la acción a requiere q unidades del recurso r”



Scheduling

Recursos consumibles

- Disponibilidad de recursos
  - Q_r Cantidad disponible del recurso r en t_0.
  - $z_r(t)$ Nivel del recurso r en el instante t.
- Producción/consumo de recursos:
  - produce(a,r,q) / consume (a,r,q) /
  - “la acción a produce/consume q unidades del recurso r”



Scheduling

Otras características de los recursos

- Recursos discretos vs. recursos continuos
- Recursos unarios ($Q_r=1$) vs. recursos múltiples ($Q_r>1$)
- Uso exclusivo vs. uso compartido (simultáneamente)
- Recursos con estados
(las acciones pueden necesitar recursos en un estado específico, p.ej. congelador a una temperatura dada).

Scheduling

Uso combinado de recursos

- Conjunción (la acción requiere el uso de múltiples recursos mientras se está realizando) .
- Disyunción (la acción requiere el uso de recursos alternativos [y su coste/duración puede variar en función del recurso utilizado]).
- Tipos/clases de recursos: $s = \{r_1,\ldots,r_m\}$
require(a,s,q)
equivale a una disyunción sobre recursos idénticos.

Scheduling

Funciones de coste y criterios de optimización

- Parámetros de la función de coste:
  - Cantidad requerida de cada tipo de recurso.
  - Tiempo durante el que se utiliza cada recurso.
- Criterios de optimización:
  - Coste total [total schedule cost]
  - Tiempo de finalización de la última tarea [makespan]
  - Tiempo de finalización ponderado
  - Número de acciones realizadas tarde [lateness]
  - Máximo retraso de una acción [tardiness]
  - Uso de recursos
  - ...

Scheduling

"Machine scheduling"

- Máquinas
  (recursos de capacidad unitaria,
  no pueden realizar dos acciones a la vez).
- Trabajos
  (conjunto parcialmente ordenado de acciones,
  cada una de las cuales requiere un tipo de recurso
  durante un número determinado de unidades de tiempo).

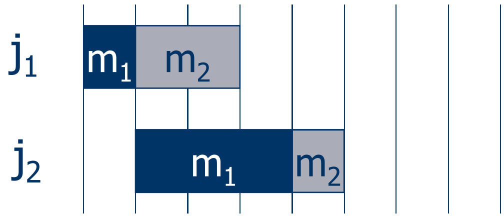
PROBLEMA: Dadas m máquinas y n trabajos, encontrar un plan [schedule] que asigne acciones a máquinas y establezca sus tiempos de inicio.

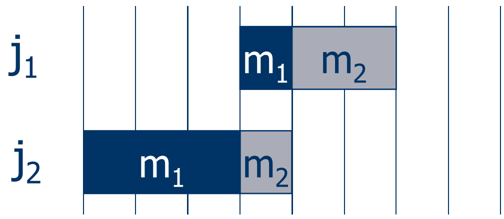
Scheduling

"Machine scheduling"
PLANIFICACIÓN POR TRABAJOS

Máquinas:
- m_1 de tipo r_1
- m_2 de tipo r_2

Trabajos:
j_1: $\langle r_1(1), r_2(2) \rangle$
j_2: $\langle r_1(3), r_2(1) \rangle$



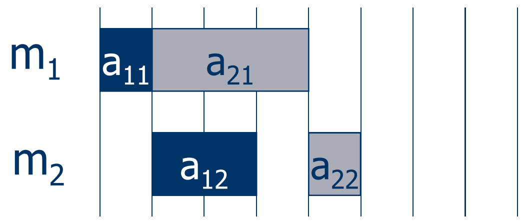


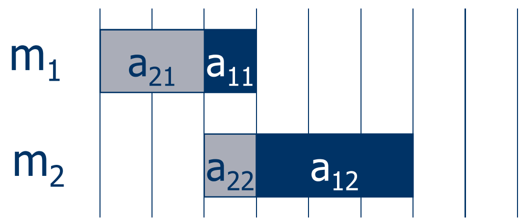
Scheduling

"Machine scheduling"
PLANIFICACIÓN POR MÁQUINAS

Máquinas:
- m_1 de tipo r_1
- m_2 de tipo r_2

Trabajos:
j_1: $\langle r_1(1), r_2(2) \rangle$
j_2: $\langle r_1(3), r_2(1) \rangle$





Notación

$\alpha \mid \beta \mid \gamma$

α Características del entorno (p.ej. máquinas).

β Características de los trabajos (tareas).

γ Criterio de optimalidad (función objetivo).

Notación

Características del entorno

$\alpha \mid \beta \mid \gamma$

1 Una sola máquina

P m máquinas idénticas en paralelo

J "job shop" (como en una cadena de montaje)
i.e. trabajos divididos en una serie de operaciones.

Notación

Características de los trabajos

$$\alpha \mid \beta \mid \gamma$$

pmtn = preemption (trabajos divisibles/interrumpibles)

$J_i \rightarrow J_k$ (relaciones de precedencia)

r_i = release dates (momento en el que están disponibles)

p_i = processing requirements (tiempo de procesamiento)

d_i = deadlines (plazos límite)

Notación

Criterio de optimalidad

$$\alpha \mid \beta \mid \gamma$$

Dos tipos de funciones de coste:

- "Cuellos de botella" max $\{f_i\}$
- Sumas Σf_i

El objetivo consiste en minimizar la función de coste…

Notación

Criterio de optimalidad

$$\alpha \mid \beta \mid \gamma$$

Fijándonos en el tiempo de finalización de cada tarea $\mathbf{C_i}$ tenemos cuatro posibles objetivos:

$\mathbf{C_{max}}$	Última tarea [makespan] = max { C_i }
$\mathbf{\Sigma C_i}$	Tiempo total [total flow time]
$\mathbf{max\ w_i C_i}$	Máximo ponderado
$\mathbf{\Sigma w_i C_i}$	Total ponderado [weighted flow time]

Notación

Criterio de optimalidad

$$\alpha \mid \beta \mid \gamma$$

Otros posibles criterios que nos pueden interesar…

$\mathbf{L_i = C_i - d_i}$	Retraso [lateness]
$\mathbf{E_i = \max\{0, d_i - C_i\}}$	Prontitud [earliness]
$\mathbf{T_i = \max\{0, C_i - d_i\}}$	Tardanza [tardiness]
$\mathbf{D_i = \lvert C_i - d_i \rvert}$	Desviación
$\mathbf{S_i = (C_i - d_i)^2}$	Desviación cuadrática

$\mathbf{U_i}$ = {**0** si $C_i \le d_i$, **1** en otro caso} = Penalización

Scheduling

Resolución de problemas de scheduling…
ALGORITMOS DE BÚSQUEDA

- Soluciones exactas
  - Algoritmos greedy
  - Programación dinámica
  - Branch & bound (NP)

- Soluciones aproximadas: Uso de heurísticas
  - Algoritmos greedy
  - Algoritmos de búsqueda local

Algoritmos greedy

Tenemos que elegir de entre un conjunto de actividades:

- Para cada actividad, conocemos su hora de comienzo y su hora de finalización.
- Podemos asistir a todas las actividades que queramos.
- Sin embargo, hay actividades que se solapan en el tiempo y no podemos estar en dos sitios a la vez.

Posibles objetivos

1. Asistir al mayor número de actividades posible.
2. Minimizar el tiempo que estamos ociosos.

Algoritmos greedy

Problema de selección de actividades

Dado un conjunto C de n actividades, con

s_i = tiempo de comienzo de la actividad i
f_i = tiempo de finalización de la actividad i

encontrar el subconjunto S de actividades compatibles de tamaño máximo (esto es, actividades que no se solapen en el tiempo).

Algoritmos greedy

Los algoritmos greedy…

- Se utilizan generalmente para resolver problemas de optimización (obtener el máximo o el mínimo).
- Toman decisiones en función de la información que está disponible en cada momento.
- Una vez tomada la decisión, ésta no vuelve a replantearse en el futuro.
- Suelen ser rápidos y fáciles de implementar.
- No siempre garantizan alcanzar la solución óptima.

Algoritmos greedy

NOTA IMPORTANTE SOBRE LOS ALGORITMOS GREEDY

El enfoque "greedy" no nos garantiza obtener soluciones óptimas.

Por lo tanto, siempre habrá que estudiar la **corrección del algoritmo** para demostrar si las soluciones obtenidas son óptimas o no.

Algoritmos greedy

Para poder resolver un problema usando el enfoque greedy, tendremos que considerar 6 elementos:

1. **Conjunto de candidatos** (elementos seleccionables).
2. **Solución parcial** (candidatos seleccionados).
3. **Función de selección** (determina el mejor candidato del conjunto de candidatos seleccionables).
4. **Función de factibilidad** (determina si es posible completar la solución parcial para alcanzar una solución del problema).
5. **Criterio que define lo que es una solución** (indica si la solución parcial obtenida resuelve el problema).
6. **Función objetivo** (valor de la solución alcanzada).

Algoritmos greedy

- Se parte de un conjunto vacío: S = ∅.
- De la lista de candidatos, se elige el mejor (de acuerdo con la función de selección).
- Comprobamos si se puede llegar a una solución con el candidato seleccionado (función de factibilidad).
  Si no es así, lo eliminamos de la lista de candidatos posibles y nunca más lo consideraremos.
- Si aún no hemos llegado a una solución, seleccionamos otro candidato y repetimos el proceso hasta llegar a una solución [o quedarnos sin posibles candidatos].

Algoritmos greedy

```
Greedy (conjunto de candidatos C): solución S

S = Ø
while (S no sea una solución y C ≠ Ø) {
      x = selección(C)
      C = C – {x}
      if (S∪{x} es factible)
         S = S∪{x}
}

if (S es una solución)
   return S;
else
   return "No se encontró una solución";
```

Selección de actividades

Elementos del algoritmo greedy

- Conjunto de candidatos: C = {actividades ofertadas}.
- Solución parcial: S (inicialmente, S=∅).
- Función de selección: menor duración, menor solapamiento, terminación más temprana…
- Función de factibilidad: x es factible si es compatible (esto es, no se solapa) con las actividades de S.
- Criterio que define lo que es una solución: C=∅.
- Función objetivo (lo que tratamos de optimizar): El tamaño de S.

Selección de actividades

Estrategias greedy alternativas

Orden en el que se pueden considerar las actividades:

- Orden creciente de hora de comienzo.
- Orden creciente de hora de finalización.
- Orden creciente de duración: $f_i - s_i$
- Orden creciente de conflictos con otras actividades (con cuántas actividades se solapa).

Selección de actividades

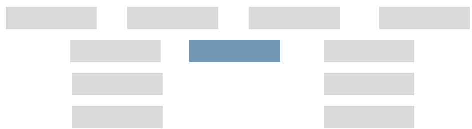
Estrategias greedy alternativas

Contraejemplos (para descartar alternativas)

- Orden creciente de hora de comienzo.

- Orden creciente de duración: f_i - s_i

- Orden creciente de conflictos con otras actividades



Algoritmos greedy
Selección de actividades

Algoritmo Greedy

`SelecciónActividades (C: actividades): S`

Ordenar C en orden creciente de tiempo de finalización.

Seleccionar la primera actividad de C

(esto es, extraerla del conjunto C y añadirla a S).

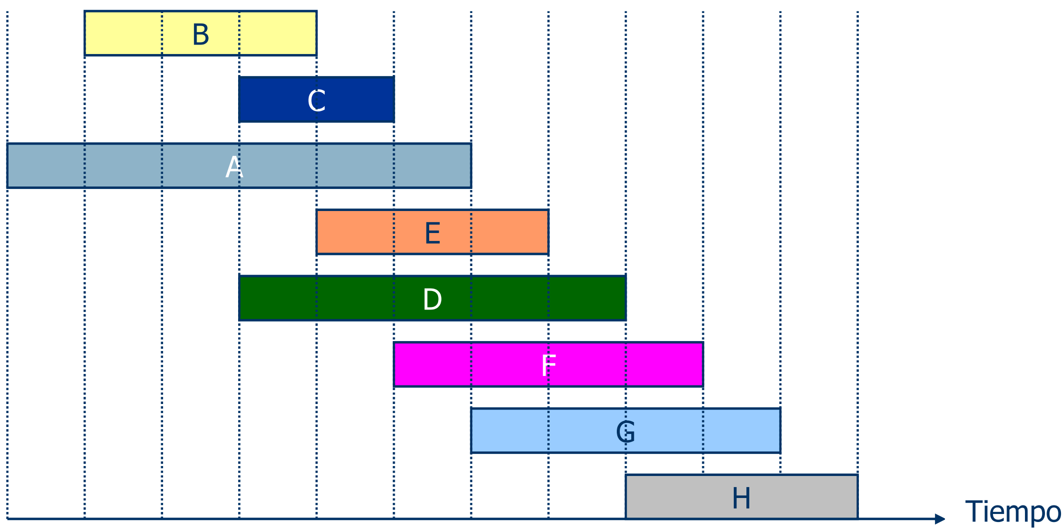
Repetir

  Extraer la siguiente actividad del conjunto ordenado:
  Si comienza después de que la actividad previa en S
  haya terminado, seleccionarla (añadirla a S).

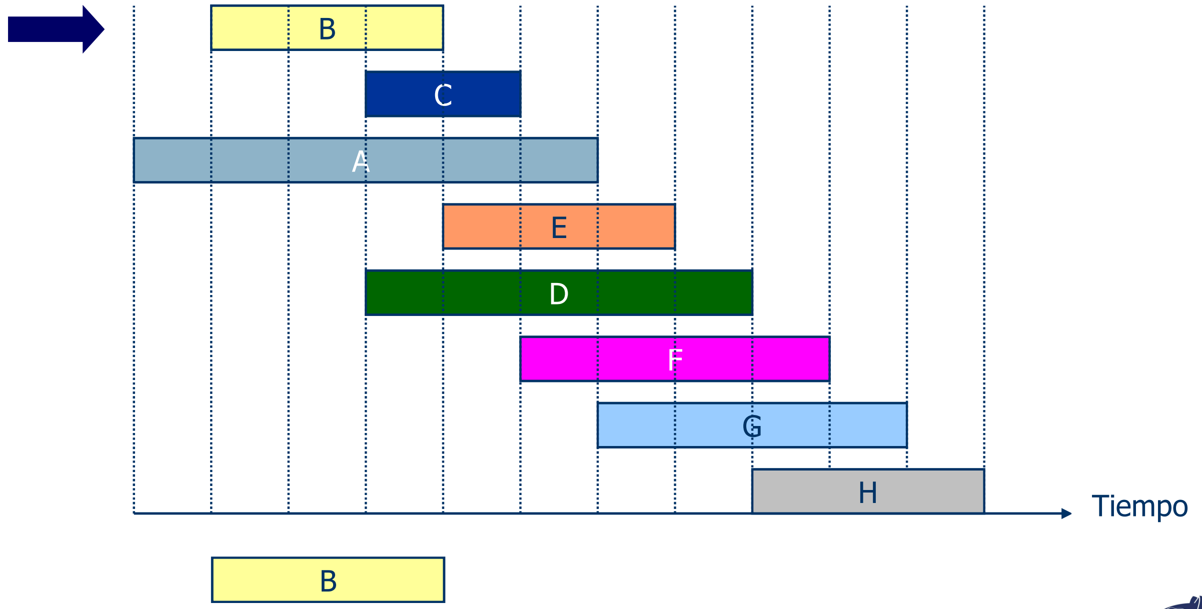
hasta que C esté vacío.

Algoritmo Greedy

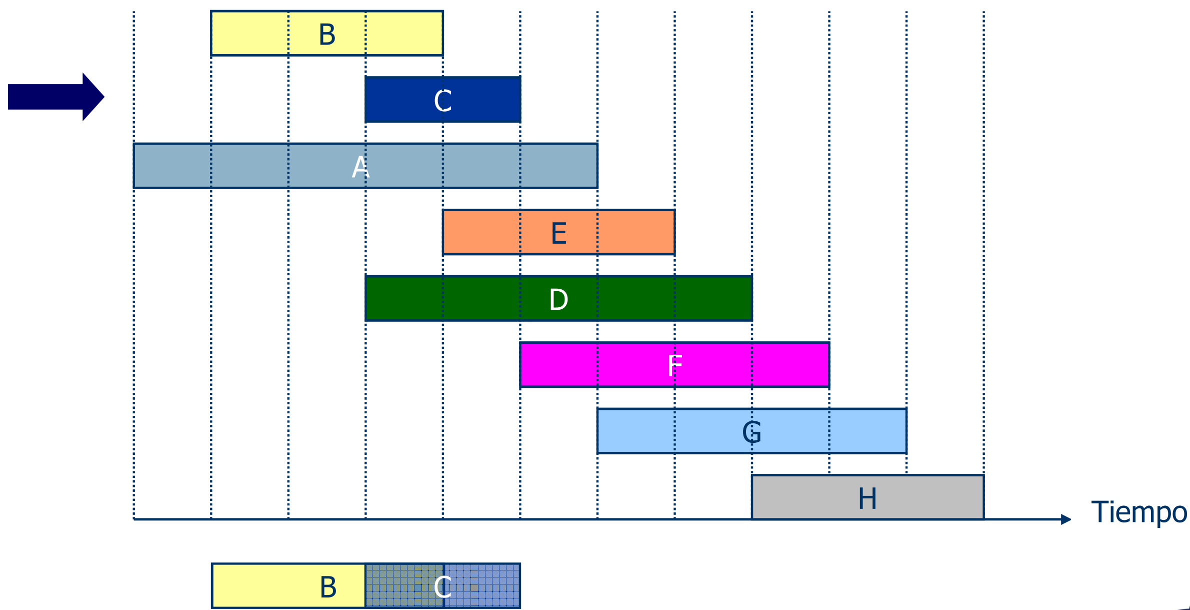
```
SelecciónActividades (C: actividades): S
{
  sort(C);      // ordenar según tiempo de finalización
  S[0] = C[0]; // seleccionar la primera actividad
  i = 1; prev = 0;
  while (i<C.length) {                // ¿solución(S)?
        x = C[i];                     // seleccionar x
        if (x.inicio >= S[prev].fin) // ¿factible(x)?
           S[++prev] = x;             // añadir x a S
        i++;
  }
}
```

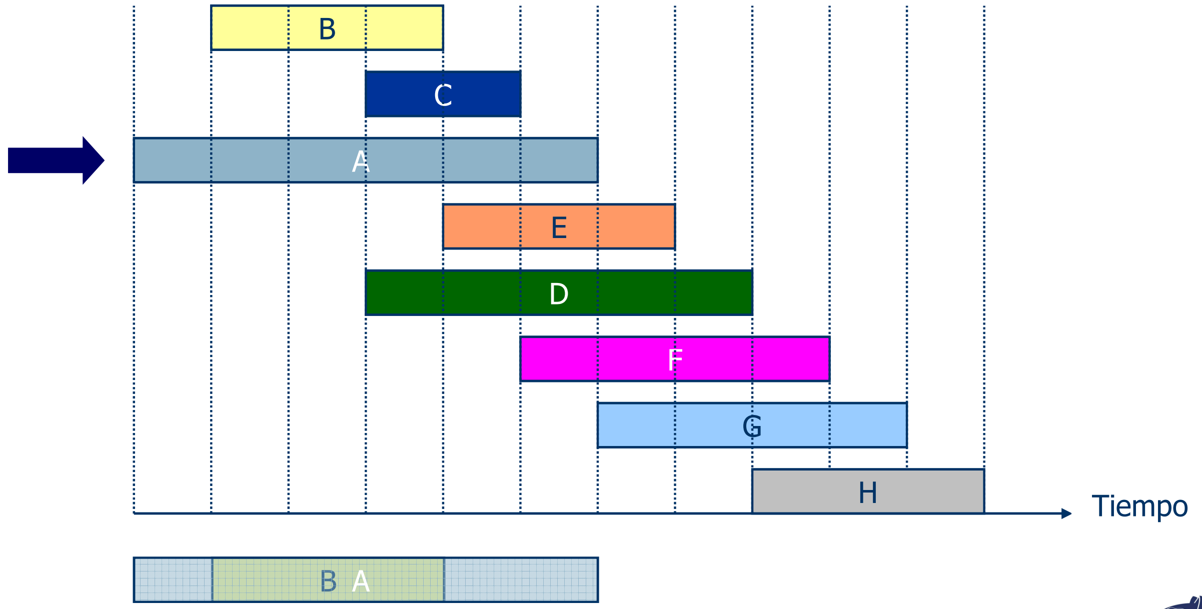
Algoritmos greedy
Selección de actividades



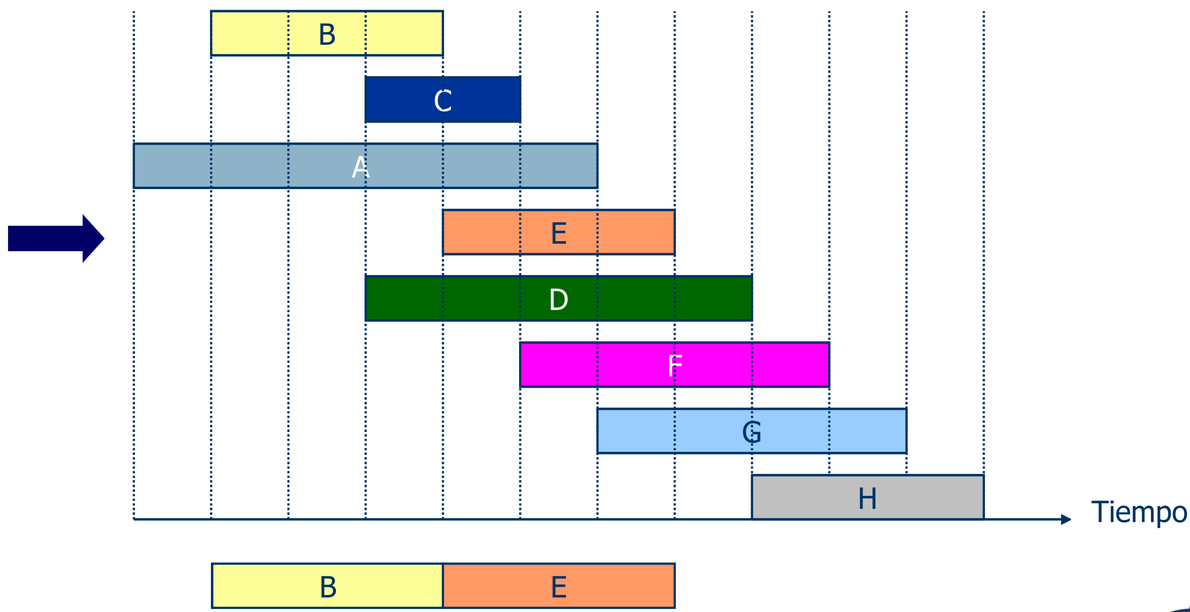
Algoritmos greedy
Selección de actividades



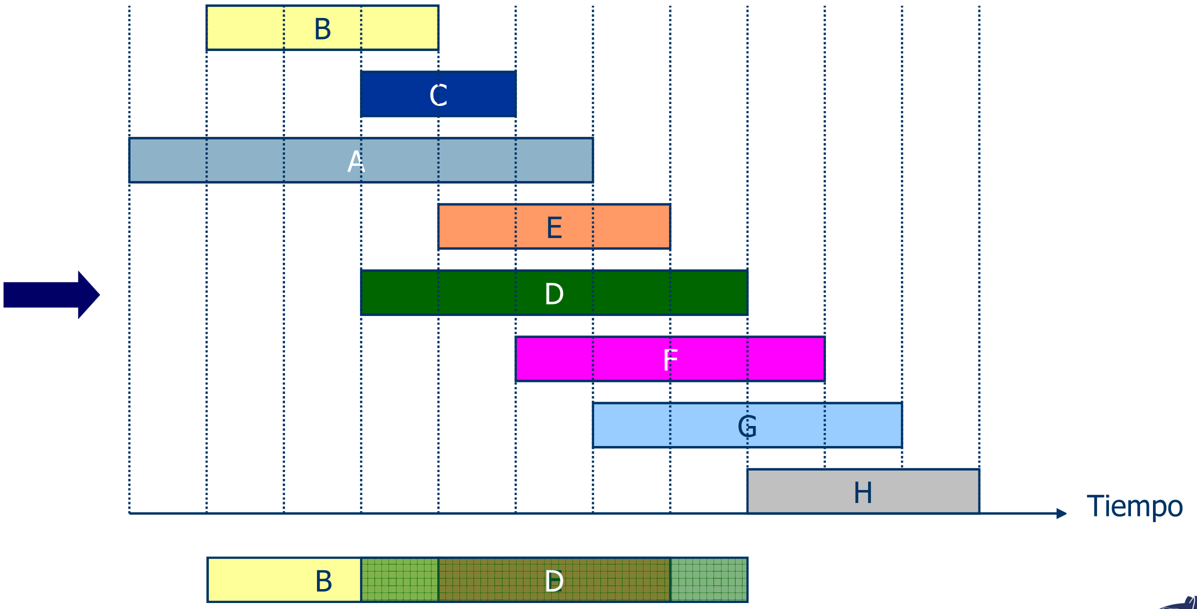
Algoritmos greedy
Selección de actividades



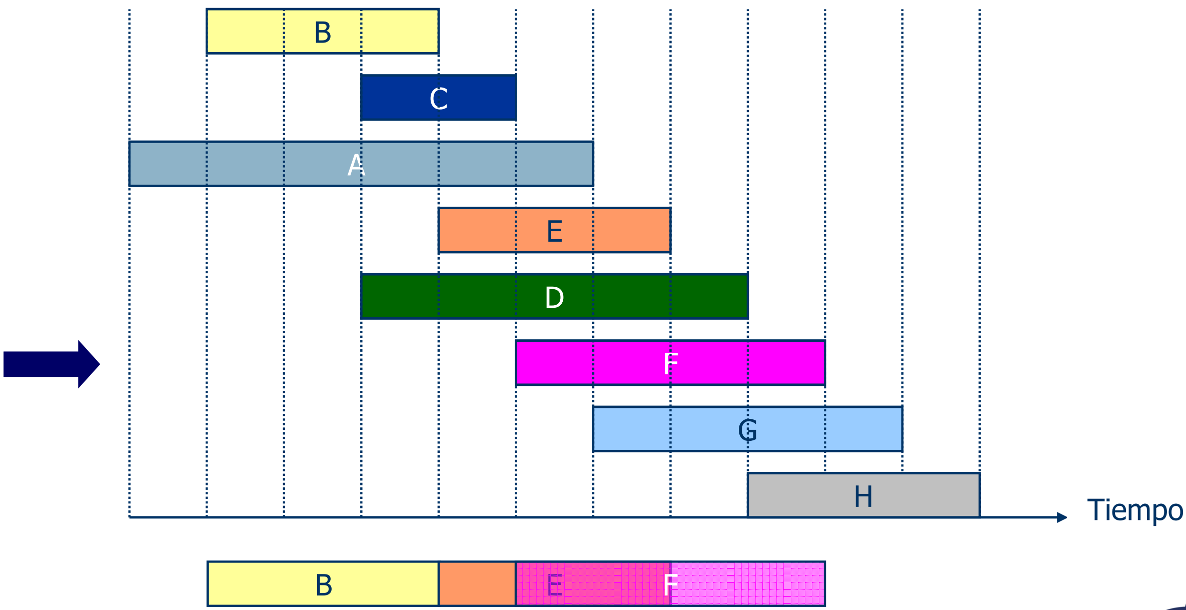
Algoritmos greedy
Selección de actividades



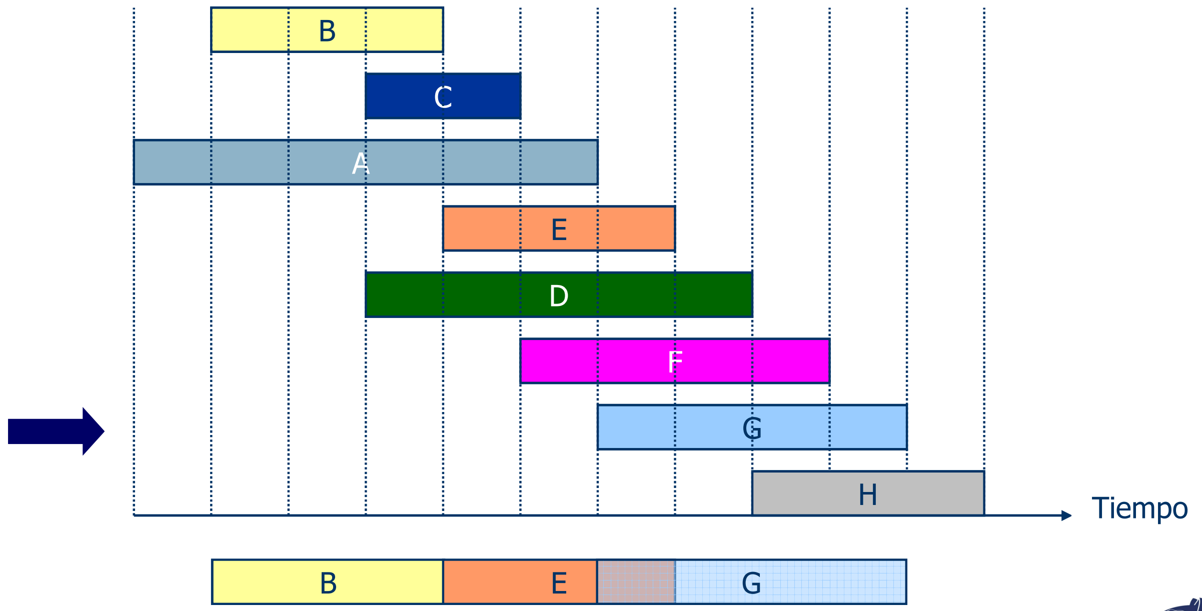
Algoritmos greedy
Selección de actividades



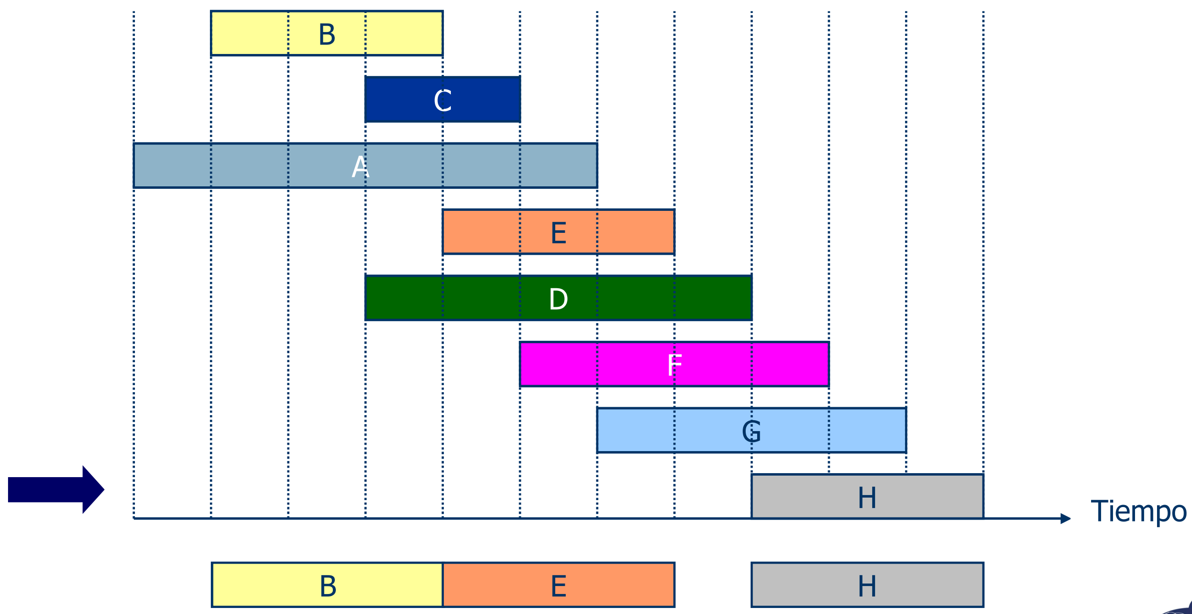
Algoritmos greedy
Selección de actividades



Algoritmos greedy
Selección de actividades

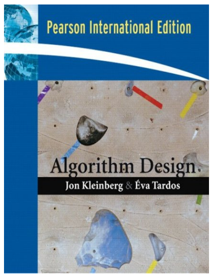


Algoritmos greedy
Selección de actividades



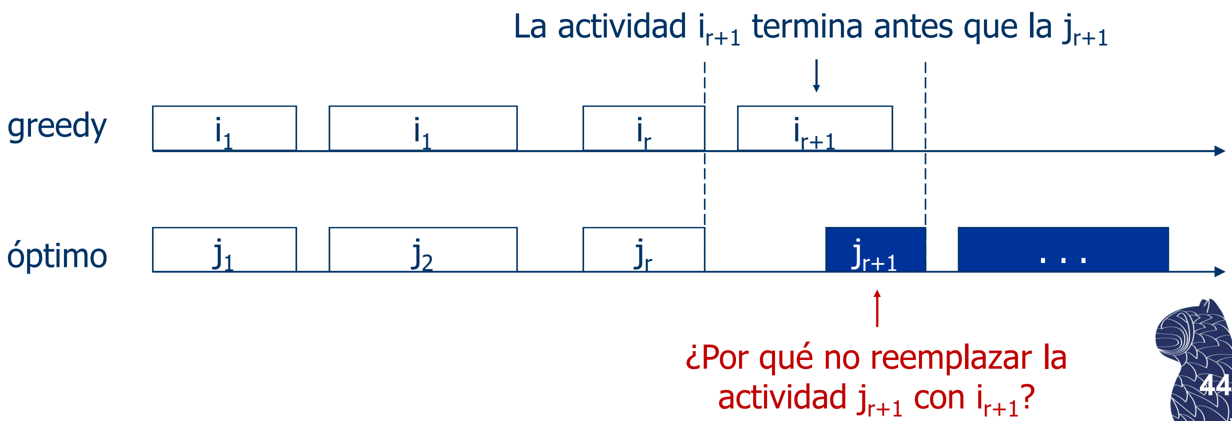
Algoritmos greedy
Selección de actividades

Demostración de optimalidad



Por reducción al absurdo:

Suponemos que el algoritmo no calcula la solución óptima…

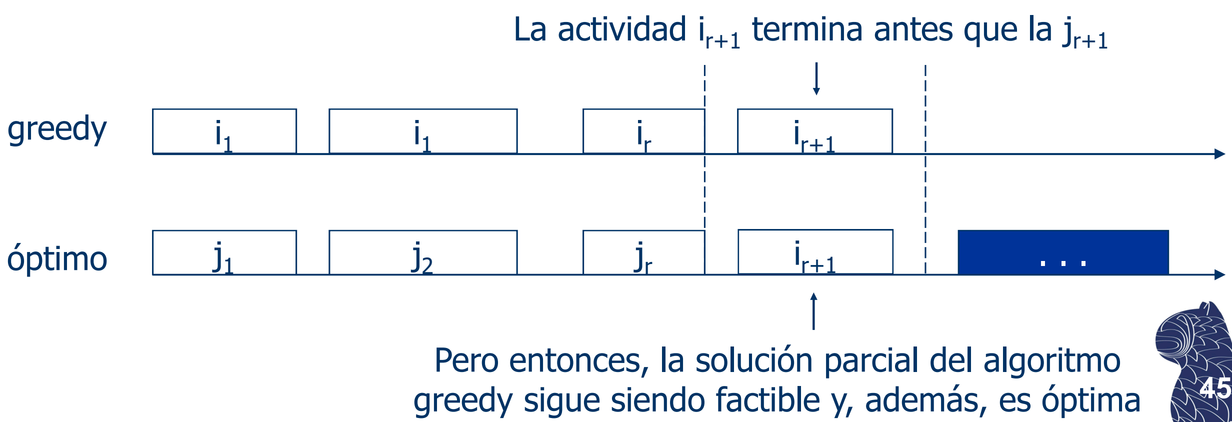


Algoritmos greedy
Selección de actividades

Demostración de optimalidad

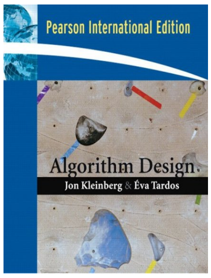
Por reducción al absurdo:

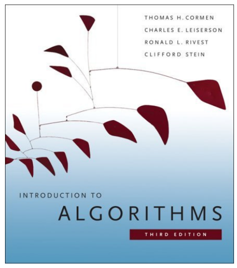
Suponemos que el algoritmo no calcula la solución óptima…



Algoritmos greedy
Selección de actividades

Demostración de optimalidad





- Jon Kleinberg & Eva Tardos: **Algorithm Design.** Sección 4.1 "Interval Scheduling: The greedy algorithm stays ahead".
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest & Clifford Stein: **Introduction to Algorithms**. [2ª edición] Sección 16.1 "An activity-selection problem".

Algoritmos greedy
Planificación de actividades

Problema $1|r_j|L_{max}$

- Problema NP en general.
- Resoluble de forma eficiente en algunos casos…

Algoritmo de Horn: O(n log n)

Si p_j=1 (todos los trabajos son de la misma duración), basta con planificar las tareas utilizando un algoritmo greedy que escoja en cada momento aquélla cuyo plazo límite está más cerca.

W.A. Horn: **Some simple scheduling algorithms**
Naval Research Logistics Quarterly 21(1):177-185, 1974

Planificación de actividades

Problema $1|r_j|L_{max}$

- Problema NP en general.
- Resoluble de forma eficiente en algunos casos…

Regla de Jackson: O(n log n)

Si r_j=r (todos los trabajos están disponibles a la vez), basta con planificar las tareas utilizando un algoritmo greedy que escoja las tareas en orden no decreciente de plazo límite [EDD = earliest due date].

Algoritmos greedy
Planificación de actividades

Problema $1|r_j|L_{max}$

- Problema NP en general.
- Resoluble de forma eficiente en algunos casos…

Otro algoritmo O(n log n)…

Si d_j=d (todos los trabajos tienen el mismo plazo límite), basta con planificar las tareas utilizando un algoritmo greedy que escoja las tareas en orden no decreciente de disponibilidad r_i.

Programación Dinámica

Esta técnica se aplica sobre problemas que presentan las siguientes características:

- **Subproblemas optimales:** La solución óptima a un problema puede ser definida en función de soluciones óptimas a subproblemas de tamaño menor.
- **Solapamiento entre subproblemas:** Al plantear la solución recursiva del problema, un mismo problema se resuelve más de una vez.

Programación Dinámica

Enfoque ascendente (bottom-up):

- Primero se calculan las soluciones óptimas para problemas de tamaño pequeño.
- Luego, utilizando dichas soluciones, encuentra soluciones a problemas de mayor tamaño.

Clave: Memorización

Almacenar las soluciones de los subproblemas en alguna estructura de datos para reutilizarlas posteriormente. De esa forma, se consigue un algoritmo más eficiente que la fuerza bruta, que resuelve el mismo subproblema una y otra vez.

Programación Dinámica

Memorización

Para evitar calcular lo mismo varias veces:

- Cuando se calcula una solución, ésta se almacena.
- Antes de realizar una llamada recursiva para un subproblema Q, se comprueba si la solución ha sido obtenida previamente:
  - Si no ha sido obtenida, se hace la llamada recursiva y, antes de devolver la solución, ésta se almacena.
  - Si ya ha sido previamente calculada, se recupera la solución directamente (no hace falta calcularla).
- Usualmente, se utiliza una matriz que se rellena conforme las soluciones a los subproblemas son calculados (espacio vs. tiempo).

Programación Dinámica

Uso de la programación dinámica:

1. Caracterizar la estructura de una solución óptima.
2. Definir de forma recursiva la solución óptima.
3. Calcular la solución óptima de forma ascendente.
4. Construir la solución óptima a partir de los datos almacenados al obtener soluciones parciales.

Estrategias de diseño

Algoritmos greedy:

Se construye la solución incrementalmente, utilizando un criterio de optimización local.

Programación dinámica:

Se descompone el problema en subproblemas solapados y se va construyendo la solución con las soluciones de esos subproblemas.

Divide y vencerás:

Se descompone el problema en subproblemas independientes y se combinan las soluciones de esos subproblemas.

Principio de Optimalidad

Para poder emplear programación dinámica, una secuencia óptima debe cumplir la condición de que cada una de sus subsecuencias también sea óptima:

Dado un problema P con n elementos,
si la secuencia óptima es $e_1e_2...e_k...e_n$ entonces:

- $e_1e_2...e_k$ es solución al problema P considerando los k primeros elementos.
- $e_{k+1}...e_n$ es solución al problema P considerando los elementos desde k+1 hasta n.

Programación Dinámica
Principio de Optimalidad

En otras palabras:

La solución óptima de cualquier instancia no trivial de un problema es una combinación de las soluciones óptimas de sus subproblemas.

- Se busca la solución óptima a un problema como un proceso de decisión "multietápico".
- Se toma una decisión en cada paso, pero ésta depende de las soluciones a los subproblemas que lo componen.

Programación Dinámica
Principio de Optimalidad

Un poco de historia: Bellman, años 50…

Enfoque está inspirado en la teoría de control: Se obtiene la política óptima para un problema de control con n etapas basándose en una política óptima para un problema similar, pero de n-1 etapas.

Etimología: Programación dinámica = Planificación temporal

En una época "hostil" a la investigación matemática, Bellman buscó un nombre llamativo que evitase posibles confrontaciones:

- "it's impossible to use dynamic in a pejorative sense"
- "something not even a Congressman could object to"

Richard E. Bellman: "Eye of the Hurricane: An Autobiography"

Principio de Optimalidad

Principio de Optimalidad de Bellman

[Bellman, R.E.: "Dynamic Programming". Princeton University Press, 1957]

"Una política óptima tiene la propiedad de que, sean cuales sea el estado inicial y la decisión inicial, las decisiones restantes deben constituir una solución óptima con respecto al estado resultante de la primera decisión".

En Informática, un problema que puede descomponerse de esta forma se dice que presenta subestructuras optimales (la base de los algoritmos greedy y de la programación dinámica).

Principio de Optimalidad

Principio de Optimalidad de Bellman

[Bellman, R.E.: "Dynamic Programming". Princeton University Press, 1957]

El principio de optimalidad se verifica si toda solución óptima a un problema está compuesta por soluciones óptimas de sus subproblemas.

¡Ojo!

El principio de optimalidad no nos dice que, si tenemos las soluciones óptimas de los subproblemas, entonces podamos combinarlas para obtener la solución óptima del problema original…

Programación Dinámica
Principio de Optimalidad

Principio de Optimalidad de Bellman

[Bellman, R.E.: "Dynamic Programming". Princeton University Press, 1957]

Ejemplo: Cambio en monedas

- La solución óptima para 0.07 euros es 0.05 + 0.02 euros.
- La solución óptima para 0.06 euros es 0.05 + 0.01 euros.
- La solución óptima para 0.13 euros **no** es (0.05 + 2) + (0.05 + 0.01) euros.

Sin embargo, sí que existe alguna forma de descomponer 0.13 euros de tal forma que las soluciones óptimas a los subproblemas nos den una solución óptima (p.ej. 0.11 y 0.02 euros).

Programación Dinámica
Definición del problema…

Para aplicar programación dinámica:

1. Se comprueba que se cumple el principio de optimalidad de Bellman, para lo que hay que encontrar la "estructura" de la solución.
2. Se define recursivamente la solución óptima del problema (en función de los valores de las soluciones para subproblemas de menor tamaño).

Programación Dinámica
… y cálculo de la solución óptima

3. Se calcula el valor de la solución óptima utilizando un enfoque ascendente:
   - Se determina el conjunto de subproblemas que hay que resolver (el tamaño de la tabla).
   - Se identifican los subproblemas con una solución trivial (casos base).
   - Se van calculando los valores de soluciones más complejas a partir de los valores previamente calculados.

4. Se determina la solución óptima a partir de los datos almacenados en la tabla.

Programación Dinámica
Selección de actividades con pesos

Enunciado del problema

Dado un conjunto C de n tareas o actividades, con

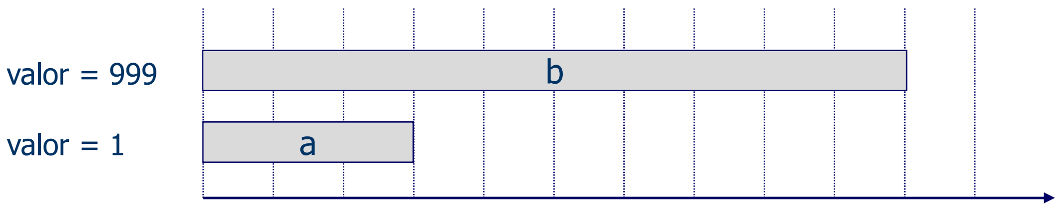
s_i = tiempo de comienzo de la actividad i
f_i = tiempo de finalización de la actividad i
v_i = valor (o peso) de la actividad i

encontrar el subconjunto S de actividades compatibles de peso máximo (esto es, un conjunto de actividades que no se solapen en el tiempo y que, además, nos proporcione un valor máximo).

Recordatorio

Existe un algoritmo greedy para este problema cuando todas las actividades tienen el mismo valor (elegir las actividades en orden creciente de hora de finalización).

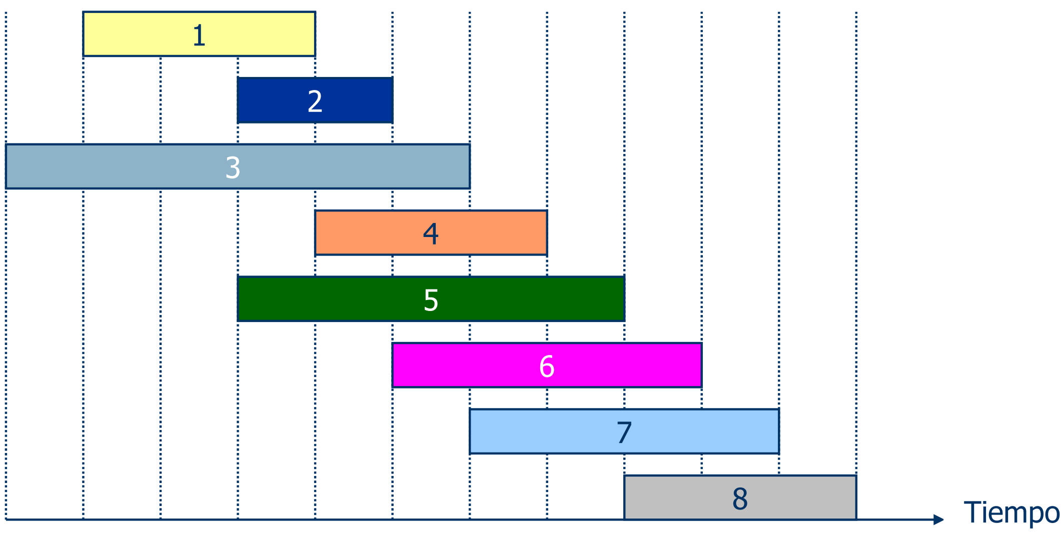
Sin embargo, el algoritmo greedy no funciona en general:



Programación Dinámica
Selección de actividades con pesos

Observación

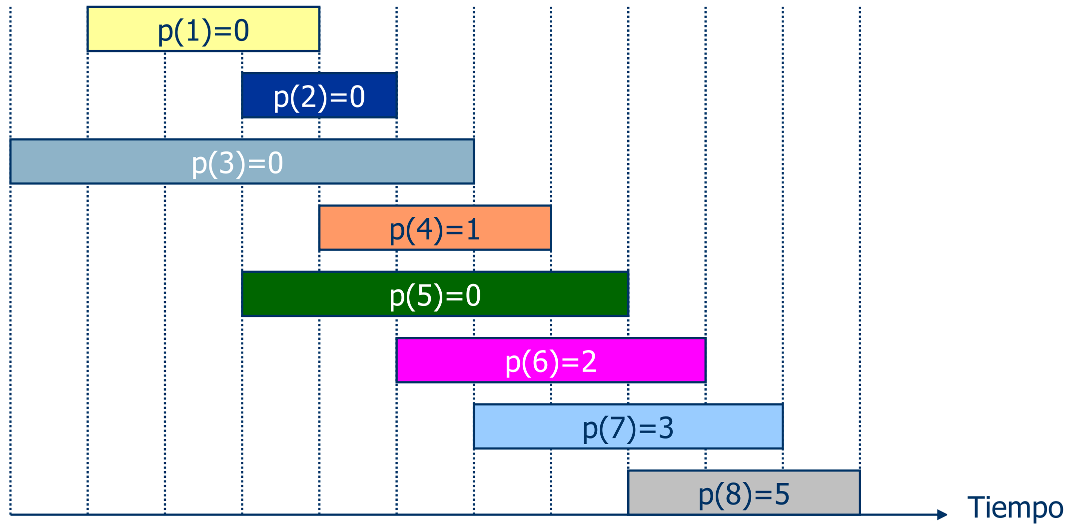
Si, como en el algoritmo greedy, ordenamos las actividades por su hora de finalización…



Programación Dinámica
Selección de actividades con pesos

Observación

… podemos definir p(j) como el mayor índice i<j
tal que la actividad i es compatible con la actividad j



Programación Dinámica
Selección de actividades con pesos

Definición recursiva de la solución

$$OPT(j) = \begin{cases} 0 & si \quad j = 0 \\ \max\{v(j) + OPT(p(j)), OPT(j-1)\} & si \quad j > 0 \end{cases}$$

- Caso 1: Se elige la actividad j.
  - No se pueden escoger actividades incompatibles >p(j).
  - La solución incluirá la solución óptima para p(j).

- Caso 2: No se elige la actividad j.
  - La solución coincidirá con la solución óptima para las primeras (j-1) actividades.

Programación Dinámica
Selección de actividades con pesos

Implementación iterativa del algoritmo

```
SelecciónActividadesConPesos (C: actividades): S
{
  Ordenar C según tiempo de finalización;      // O(n log n)
  Calcular p[1]..p[n];                         // O(n log n)

  mejor[0] = 0;                                // O(1)
  for (i=1; i<=n, i++)                         // O(n)
      mejor[i] = max ( valor[i]+mejor[p[i]],
                       mejor[i-1] );

  return Solución(mejor);                      // O(n)
}
```


Programación Dinámica
Planificación de actividades con pesos

Algunos problemas de secuenciación de tareas se pueden resolver utilizando programación dinámica:

- Problema $1||\Sigma w_i U_i$
- Problema $P||\Sigma w_i C_i$

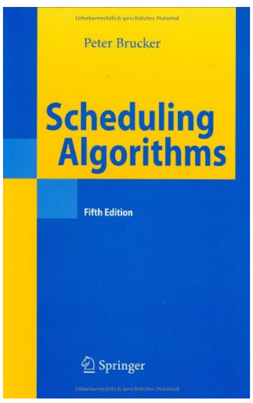
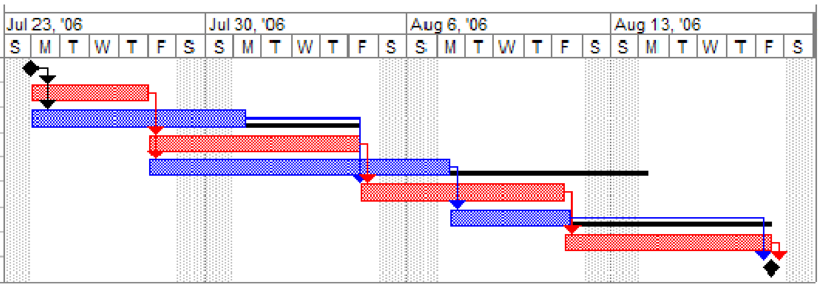


Diagrama de Gantt

ID	Task Name	Predecessors	Duration
1	Start		0 days
2	a	1	4 days
3	b	1	5.33 days
4	c	2	5.17 days
5	d	2	6.33 days
6	e	3,4	5.17 days
7	f	5	4.5 days
8	g	6	5.17 days
9	Finish	7,8	0 days



Microsoft Project

Programación Dinámica
Planificación de proyectos

Camino crítico

Conjunto de tareas que deben completarse de acuerdo al plan si queremos que el proyecto en su conjunto termine en la fecha establecida.

Los métodos de planificación de proyectos proporcionan herramientas para determinar el camino crítico:

- **CPM** [Critical Path Method]
- **PERT** [Program Evaluation and Review Technique]

Programación Dinámica
Planificación de proyectos: CPM

Dadas las duraciones de cada tarea
y las dependencias existentes entre ellas:

- **ES** [Earliest Start]: Comienzo más temprano
  $\mathbf{ES(t) = max_{p \rightarrow t} \{ ES(p) + duración(p) \}}$
  siendo ES(t)=0 para las tareas sin predecesoras.

- **LS** [Latest Start]: Comienzo más tardío
  $\mathbf{LS(t) = min_{s \leftarrow t} \{ LS(s) - duración(t) \}}$
  siendo LS(t)=ES(t) para las tareas sin sucesoras.

James Kelley (Remington Rand) & Morgan Walker (DuPont)
"Critical-Path Planning and Scheduling"
Proceedings of the Eastern Joint Computer Conference, 1959.

Programación Dinámica
Planificación de proyectos: CPM

Dadas las duraciones de cada tarea
y las dependencias existentes entre ellas:

- **Holgura [slack, a.k.a. float]**
  $\mathbf{slack(t) = LS(t) - ES(t)}$

- **Actividades críticas**:
  Tareas sin holgura.

Early Start	Duration	Early Finish
	Task Name	
Late Start	Slack	Late Finish

Programación Dinámica
Planificación de proyectos: CPM

Camino crítico

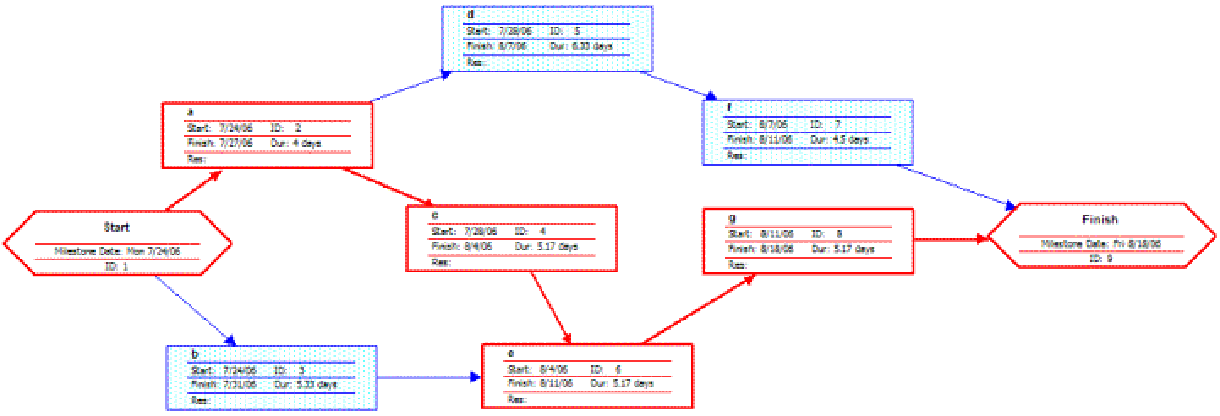


Diagrama AON [activity on node]
Microsoft Project

Branch & Bound

B&B es una generalización de la técnica de backtracking:

- Se realiza un recorrido sistemático del árbol de estados de un problema, si bien ese recorrido no tiene por qué ser en profundidad, como sucedía en backtracking: usaremos una **estrategia de ramificación**.
- Además, utilizaremos **técnicas de poda** para eliminar todos aquellos nodos que no lleven a soluciones óptimas (estimando, en cada nodo, cotas del beneficio que podemos obtener a partir del mismo).

NOTA: Los algoritmos que utilizan B&B suelen ser de orden exponencial (o peor) en su peor caso.

Branch & Bound

Diferencias con backtracking

- En backtracking, tan pronto como se genera un nuevo hijo, este hijo pasa a ser el nodo actual (en profundidad).
- En B&B, se generan todos los hijos del nodo actual antes de que cualquier otro nodo vivo pase a ser el nuevo nodo en curso (**no** se realiza un recorrido en profundidad).

En consecuencia:

- En backtracking, los únicos nodos vivos son los que están en el camino de la raíz al nodo en curso.
- En B&B, puede haber más nodos vivos, que se almacenan en una lista de nodos vivos.

Branch & Bound

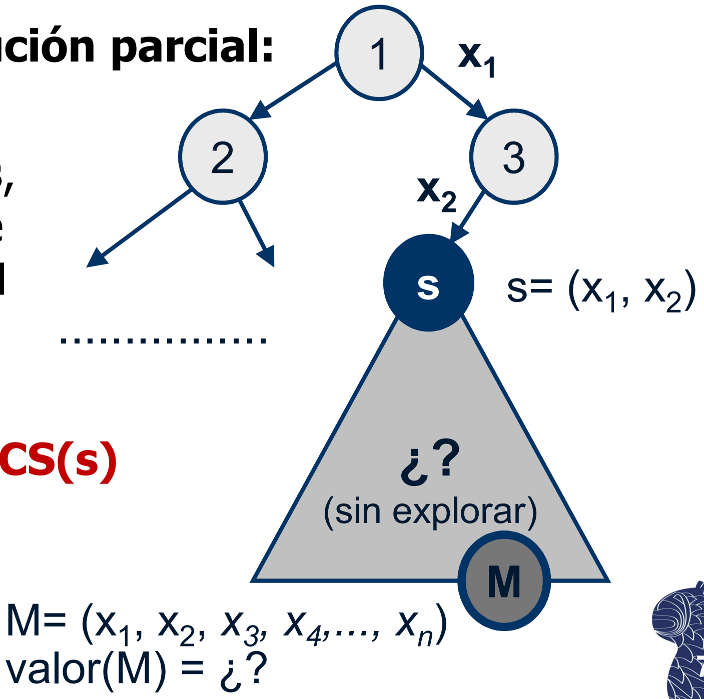
Diferencias con backtracking

- En backtracking, el test de comprobación realizado por la funciones de poda nos indica únicamente si un nodo concreto nos puede llevar a una solución o no.
- En B&B, sin embargo, se acota el valor de la solución a la que nos puede conducir un nodo concreto, de forma que esta acotación nos permite…
  - podar el árbol (si sabemos que no nos va a llevar a una solución mejor de la que ya tenemos) y
  - establecer el orden de ramificación (de modo que comenzaremos explorando las ramas más prometedores del árbol).

Branch & Bound

Estimación de las cotas a partir de una solución parcial:

Antes de explorar **s**, se acota el valor de la mejor solución **M** alcanzable desde **s**.



CI(s) ≤ valor(M) ≤ CS(s)

Branch & Bound

Estimadores y cotas en Branch & Bound

	Problema de maximización	Problema de minimización
Valor	Beneficio	Coste
Cota local	Cota superior	Cota inferior
	CL ≥ Óptimo local	CL ≤ Óptimo local
	Interpretación: No alcanzaremos nada mejor al expandir el nodo.	
Cota global	Cota inferior	Cota superior
	CG ≤ Óptimo global	CG ≥ Óptimo global
	Interpretación: La solución óptima nunca será peor que esta cota.	

Branch & Bound

Estrategia de poda en Branch & Bound

Además de podar aquellos nodos que no cumplan las restricciones del problema (soluciones parciales no factibles), **se podrán podar aquellos nodos cuya cota local sea peor que la cota global**.

Si sé que lo mejor que se puede alcanzar al expandir un nodo no puede mejorar una solución que ya se ha obtenido (o se va a obtener al explorar otra rama del árbol), no es necesario expandir dicho nodo.

Branch & Bound

Estrategia de poda en Branch & Bound

	Problema de maximización	**Problema de minimización**
Valor	Beneficio	Coste
Podar si…	**CL < CG**	**CL > CG**
Cota local	CL ≥ Óptimo local	CL ≤ Óptimo local
	Interpretación: No alcanzaremos nada mejor al expandir el nodo.	
Cota global	CG ≤ Óptimo global	CG ≥ Óptimo global
	Interpretación: La solución óptima nunca será peor que esta cota.	

Técnicas heurísticas

Acciones asignables

Sea S un plan parcialmente definido para un problema P:

- Una acción a_{ji} de un trabajo j_l no está asignada si no aparece en S.
- Una acción a_{ji} de un trabajo j_l es asignable si no tiene predecesores no asignados en S.

Técnicas heurísticas

Acciones asignables

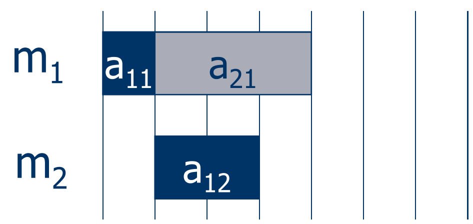
Máquinas

- m_1 de tipo r_1
- m_2 de tipo r_2

Trabajos

- j_1: $\langle r_1(1), r_2(2)\rangle$
- j_2: $\langle r_1(3), r_2(1)\rangle$
- j_2: $\langle r_1(3), r_2(1), r_1(3)\rangle$

Plan parcial S:



No asignadas

a_{22}, a_{31}, a_{32}, a_{33}

Asignables

a_{22}, a_{31}

Técnicas heurísticas

Una posible heurística:

EAT [Earliest Assignable Time]

"Tiempo asignable más temprano"

El EAT de una acción asignable a_{ji} a la máquina m en el plan parcial S es el máximo de

- el tiempo de finallización de la última acción asignada a m en S, y
- el tiempo de finalización del último predecesor (a_{j0} … a_{ji-1}) de a_{ji} en S.

¡OJO! La asignación no tiene por qué ser óptima.

Técnicas heurísticas

EAT [Earliest Assignable Time]

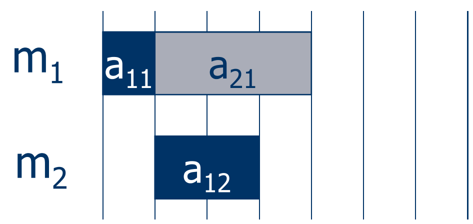
Máquinas

- m_1 de tipo r_1
- m_2 de tipo r_2

Trabajos

- j_1: $\langle r_1(1), r_2(2)\rangle$
- j_2: $\langle r_1(3), r_2(1)\rangle$
- j_2: $\langle r_1(3), r_2(1), r_1(3)\rangle$

Plan parcial S:



$EAT(a_{22,}\ m_2) = 4$

$EAT(a_{32,}\ m_1) = 4$

Técnicas heurísticas

Planificador heurístico EAT [greedy]

heuristicScheduler(P,S)
  assignables ← P.getAssignables(S)
  if assignables.isEmpty() **then return** S
  action ← assignables.selectOne()
  machines ← P.getMachines(action)
  machine ← machines.selectOne()
  time ← S.getEarliestAssignableTime(action, machine)
  S ← S + assign(action, machine, time)
  return heuristicScheduler(P,S)

Técnicas heurísticas

Técnicas de búsqueda local

- Generar un plan inicial aleatorio.

Repetir

- Generar vecinos del plan actual

  p.ej. cambiar la máquina asignada a una acción
  cambiar la posición de una acción en el plan

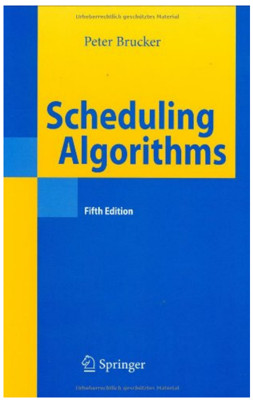
- Evaluar vecinos
  (aplicando el criterio de optimización del problema).

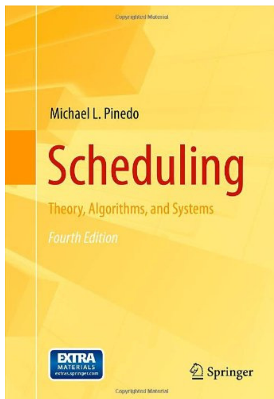
Técnicas heurísticas

Planificador basado en técnicas de búsqueda local

```
function LocalSearchScheduler(P)
best ← randomSchedule(P)
loop MAXLOOP times
    S ← randomSchedule(P)
    do
        succs ← S.getBestNeighbours(P)
        next ← succs.selectOne()
        if S.evaluate() < next.evaluate() then
            S ← next
    while S = next
    if S.evaluate() > best.evaluate() then
        best ← S
return best
```


Bibliografía

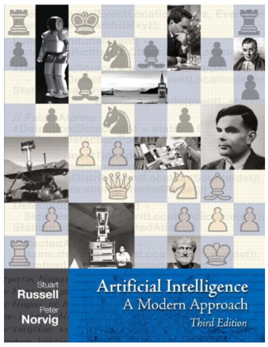




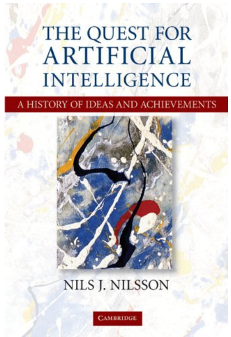
- Peter Brucker:
  Scheduling Algorithms
  Springer, 5th edition, 2007
  ISBN 354069515X
- Michael L. Pinedo:
  Scheduling: Theory, Algorithms, and Systems
  Springer, 4th edition, 2012
  ISBN 1461419867
  http://www.stern.nyu.edu/om/faculty/pinedo/schedtheory

Bibliografía

- Stuart Russell & Peter Norvig:
  Artificial Intelligence:
  A Modern Approach
  Prentice-Hall, 3rd edition, 2009
  ISBN 0136042597
  http://aima.cs.berkeley.edu/



- Nils J. Nilsson:
  The Quest for Artificial Intelligence
  Cambridge University Press, 2009
  ISBN 0521122937
  http://ai.stanford.edu/~nilsson/QAI/qai.pdf



Bibliografía

Enlaces de interés

- **Complexity results** (Universität Osnabrück)
  http://www.informatik.uni-osnabrueck.de/knust/class/
- **TORSCHE** (Czech Technical University, Prague)
  http://rtime.felk.cvut.cz/scheduling-toolbox/
- **MSC Automated Planning** (University of Edinburgh)
  http://www.inf.ed.ac.uk/teaching/courses/plan/
  https://www.coursera.org/course/aiplan
- **CS541 Artificial Intelligence Planning** (USC)
  http://www.isi.edu/~blythe/cs541/