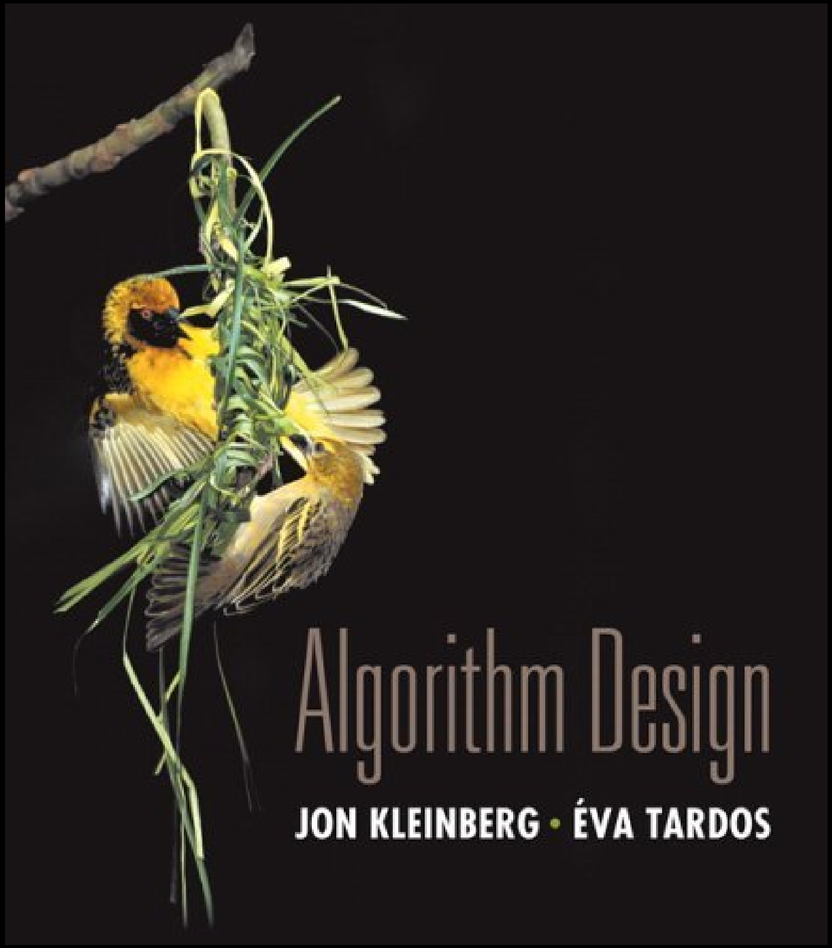


Chapitre 4

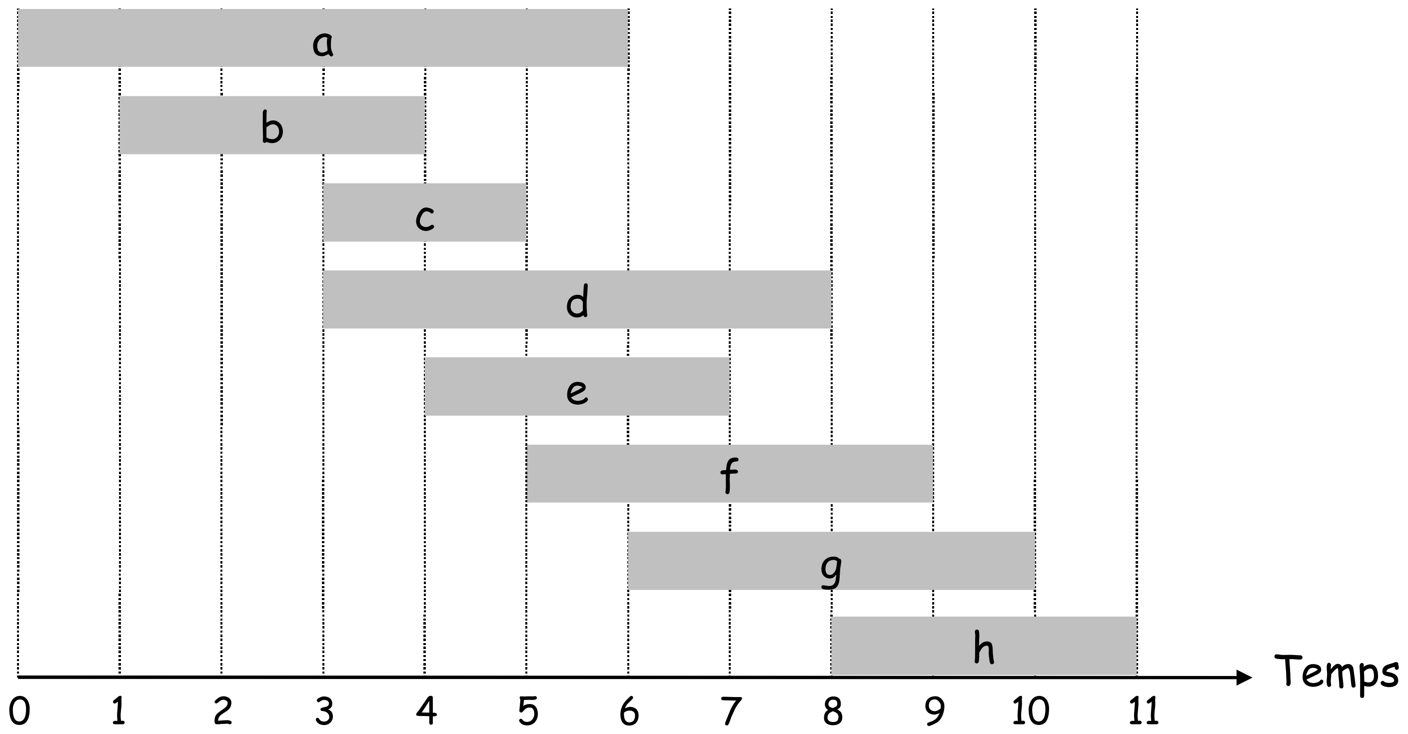
Algorithmes Gloutons

Interval Scheduling

Interval Scheduling

Interval scheduling.

- La tâche j commence à l'instant s_j et finit à l'instant f_j.
- Deux tâches sont compatibles si elles ne se chevauchent pas.
- But : trouver un ensemble maximal de tâches compatibles.



Interval Scheduling : Algorithmes gloutons

Approche gloutonne. On considère les tâches dans un certain ordre. On retient chaque tâche à condition qu'elle soit compatible avec les tâches déjà retenues.

- [Earliest start time] On considère les tâches dans l'ordre chronologique de leur commencement (ordre croissant des s_j.)

- [Earliest finish time] On considère les tâches dans l'ordre chronologique de leur fin (ordre croissant des f_j.)

- [Shortest interval] On considère les tâches dans l'ordre chronologique de leur durée (ordre croissant des $f_j - s_j$.)

- [Fewest conflicts] Pour chaque tâche, on compte le nombre c_j de tâches avec lesquelles elle entre en conflit. On considère les tâches selon l'ordre croissant des c_j.

Interval Scheduling : Algorithmes gloutons

Approche gloutonne. On considère les tâches dans un certain ordre. On retient chaque tâche à condition qu'elle soit compatible avec les tâches déjà retenues.

Ruine le critère 'earliest start time'

Ruine le critère 'shortest interval'

Ruine le critère 'fewest conflicts'

Interval Scheduling : Algorithmes gloutons

Approche gloutonne. On considère les tâches dans un certain ordre. On retient chaque tâche à condition qu'elle soit compatible avec les tâches déjà retenues.

```
Trier les tâches selon l'ordre chronologique de leur
fin, de sorte que f1 ≤ f2 ≤ ... ≤ fn.
   tâches sélectionnées
A ← φ
pour j = 1 à n {
   si (tâche j compatible avec A)
      A ← A ∪ {j}
}
retourner A
```

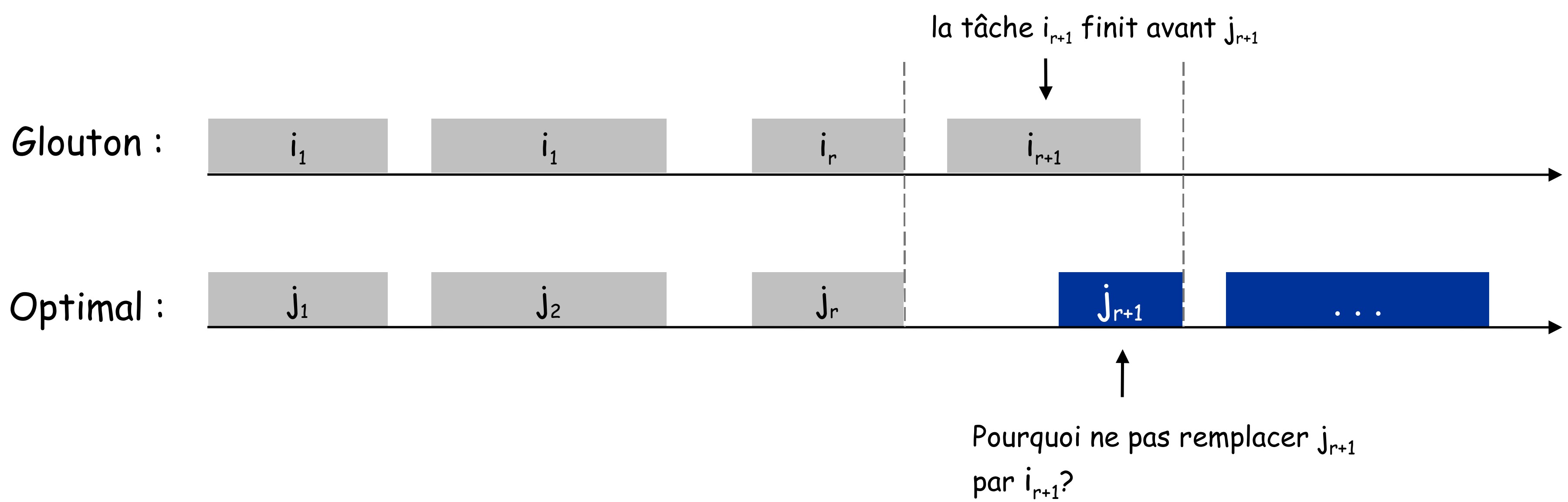
Implémentation. $O(n \log n)$.

- Soit j^* la dernière tâche ajoutée à A.
- La tâche j est compatible avec A si $s_j \geq f_{j^*}$.

Théorème. L'algorithme glouton 'earliest finish time' est optimal.

Preuve. (par l'absurde)

- On suppose que l'algorithme n'est pas optimal.
- Soit $i_1, i_2, \ldots i_k$ l'ensemble des tâches sélectionnées par l'algorithme.
- Soit $j_1, j_2, \ldots j_m$, $m>k$, un ensemble maximal ordonné selon l'ordre chronologique des fins de tâches ; soit r, $r<k$, le plus grand entier naturel tel que $i_1 = j_1, i_2 = j_2, \ldots, i_r = j_r$.

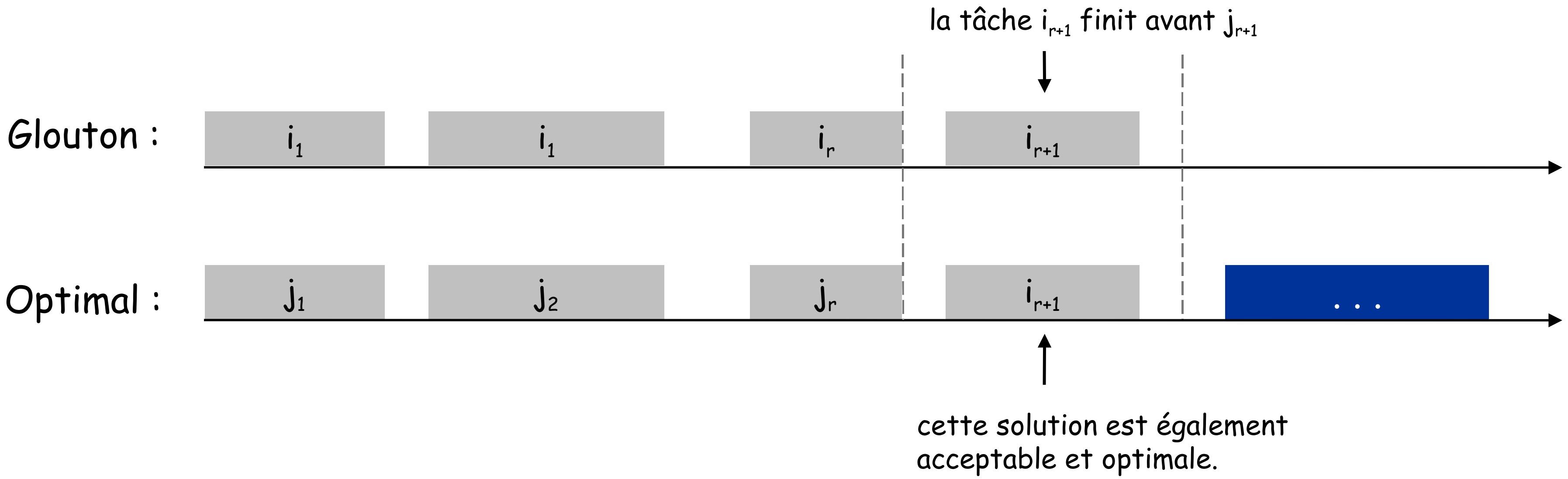


Interval Scheduling : *Analyse*

Théorème. L'algorithme glouton 'earliest finish time' est optimal.

Preuve. (par l'absurde)

- On montre que i_1, i_2, ... i_r,i_{r+1}, j_{r+2}, ... j_m est maximal ;
- En itérant le raisonnement, on montre que i_1, i_2, ... i_k , j_{k+1}, ... j_m est maximal.
- Conclusion : ou bien m=k (et i_1, i_2, ... i_k est maximal) ou bien i_1, i_2, ... i_k n'est pas l'output de l'algorithme.



Partitionnement d'intervalle

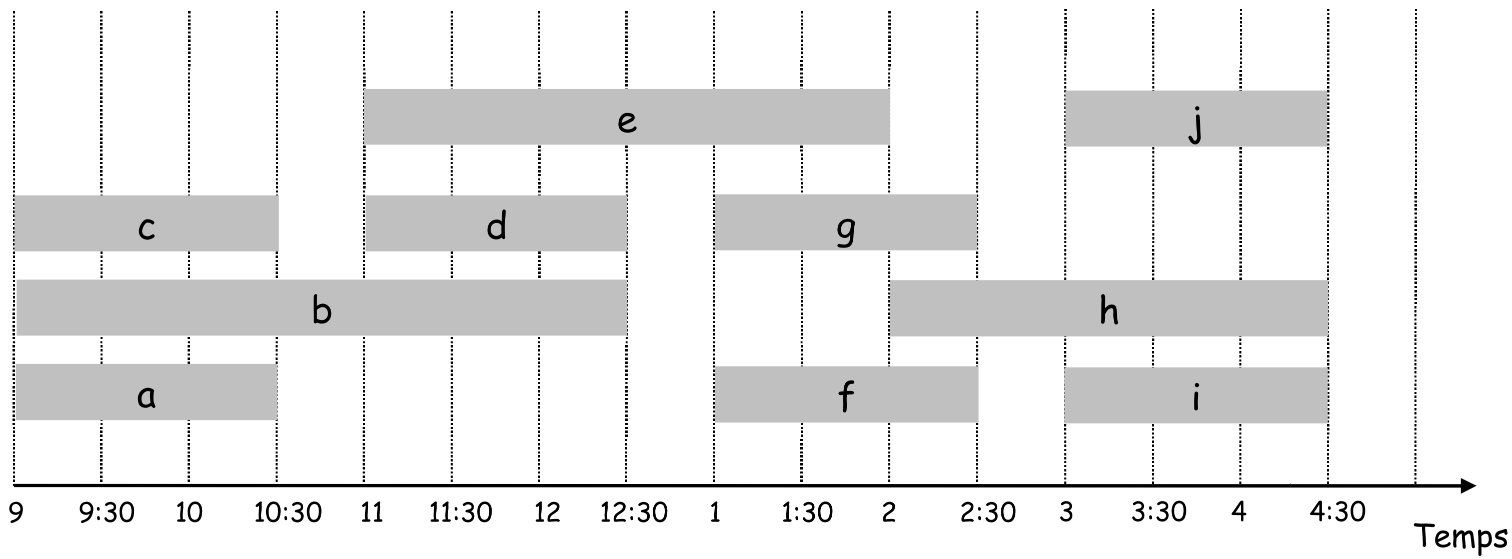
Interval Partitioning

Interval Partitioning

Interval partitioning.

- Le cours j commence à l'instant s_j et finit à l'instant f_j.
- But : trouver un nombre minimal de salles permettant de programmer les cours dans des salles distinctes.

Ex: 4 salles nécessaires pour programmer ces 10 cours.

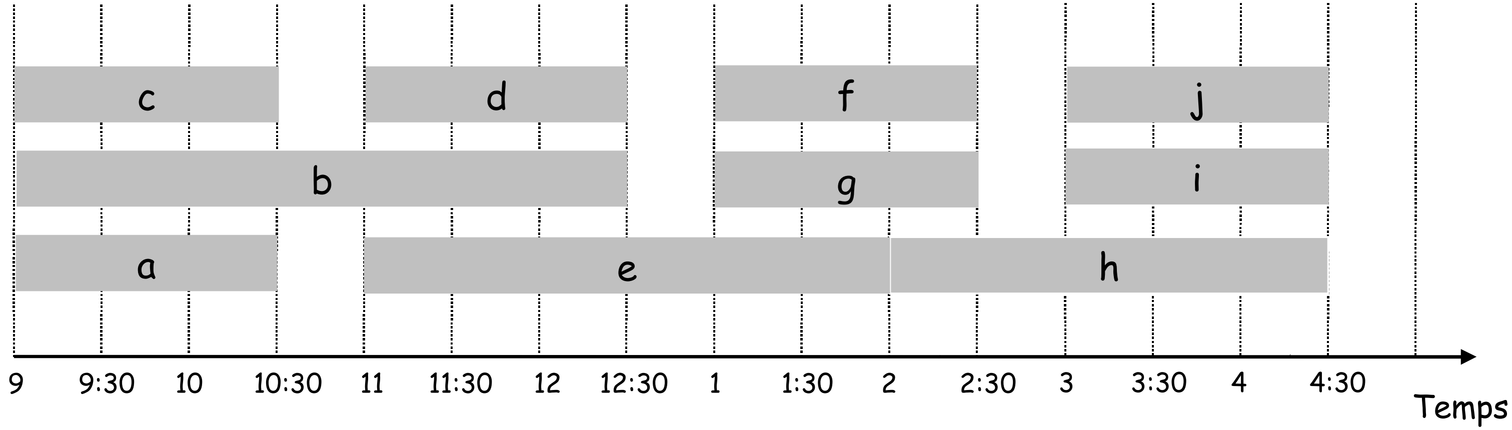


Interval Partitioning

Interval partitioning.

- Le cours j commence à l'instant s_j et finit à l'instant f_j.
- But : trouver un nombre minimal de salles permettant de programmer les cours dans des salles distinctes.

Ex: Dans ce cas seulement 3 salles sont nécessaires.




3:30

Interval Partitioning : Algorithme glouton

Algorithme glouton. On considère les cours dans l'ordre chronologique de leur début : on attribue une salle compatible à chaque cours.

```
Trier les intervalles par ordre chronologique de leur fin
de sorte que s₁ ≤ s₂ ≤ ... ≤ sₙ.
d ← 0
    Nombre de salles utilisées

pour j = 1 à n {
   si (le cours j est compatible avec une salle, k)
      programmer le cours j dans la salle k
   sinon
      utiliser une nouvelle salle d + 1
      programmer le cours j dans la salle d + 1
      d ← d + 1
}
```

Implémentation. $O(n \log n)$.

- Pour chaque salle k, on garde trace de l'instant de fin du dernier cours qui lui a été affecté.
- On garde les salles dans une file de priorité (priority queue.)

Interval Partitioning : Analyse de l'approche gloutonne

Observation. L'algorithme glouton ne programme jamais deux cours incompatibles dans la même salle.

Théorème. L'algorithme glouton est optimal.

Preuve.

- Soit d = nombre de salles que l'algorithme glouton affecte.
- La salle d a été ouverte car il était nécessaire de programmer un cours, par exemple j, qui était incompatible avec les d-1 autres salles.
- Puisqu'on trie selon l'ordre chronologique des débuts de cours, toutes ces incompatibilités étaient causées par des cours dont le début n'est pas postérieur à s_j.
- On a donc d cours qui se chevauchent à l'instant $s_j + \varepsilon$, pour un certain $\varepsilon > 0$.
- Observation clé $\Rightarrow$ aucune programmation ne peut utiliser moins de d salles. ▪

Minimiser le retard

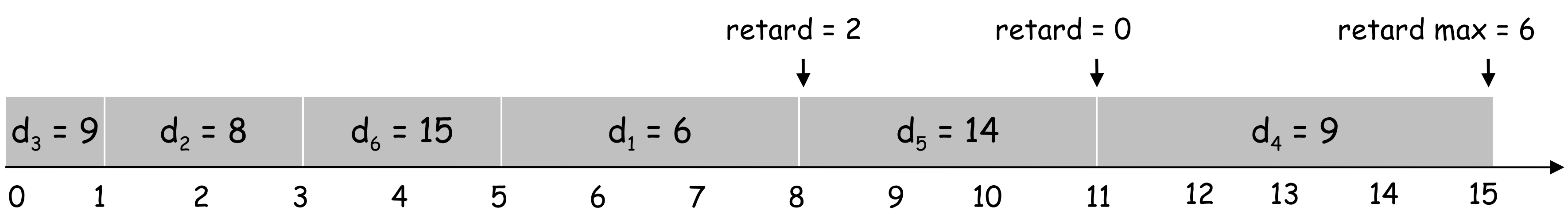
Organiser les tâches afin de minimiser le retard

Minimizing lateness problem.

- Un processeur (unique) exécute une tâche (et une seule) à la fois.
- L'exécution de la tâche j demande t_j unités de temps de travail du prcesseur et doit être achevée à l'instant d_j.
- Si la tâche j commence à l'instant s_j, elle finit à l'instant $f_j = s_j + t_j$.
- Retard : $\quad_j = \max \{ 0, \; f_j - d_j \}$.
- But : programmer les tâches afin de minimiser le retard maximal L = max $\quad_j$.

Ex:

	1	2	3	4	5	6
t_j	3	2	1	4	3	2
d_j	6	8	9	9	14	15



Minimizing Lateness : algorithmes gloutons

"Gabarit" glouton. On traite les tâches dans un certain ordre.

- [Shortest processing time first] On considère les tâches dans l'ordre croissant de leur durée d'exécution t_j.

- [Earliest deadline first] On considère les tâches dans l'ordre chronologique de leur instant-limite d_j.

- [Smallest slack] On considère les tâches dans l'ordre croissant des différences $d_j - t_j$.

Minimizing Lateness : algorithmes gloutons

"Gabarit" glouton. On traite les tâches dans un certain ordre.

- [Shortest processing time first] On considère les tâches dans l'ordre croissant de leur durée d'exécution t_j.

	1	2
t_j	1	10
d_j	100	10

contre-exemple

- [Smallest slack] On considère les tâches dans l'ordre croissant des différences $d_j - t_j$.

	1	2
t_j	1	10
d_j	2	10

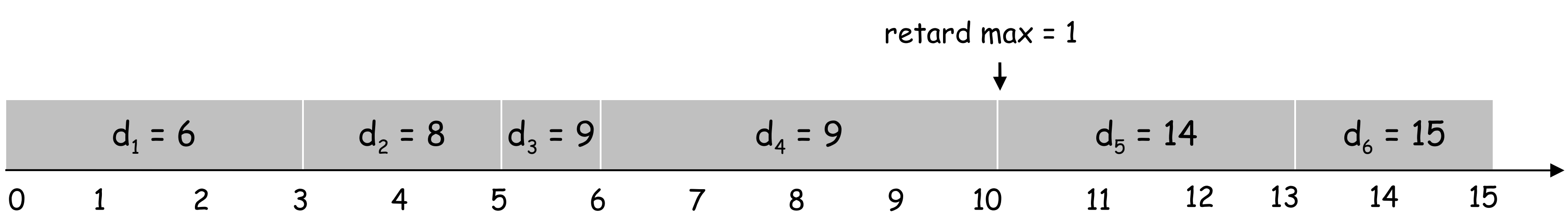
contre-exemple

Minimizing Lateness : algorithmes gloutons

Algorithme glouton. Stratégie "Earliest deadline first".

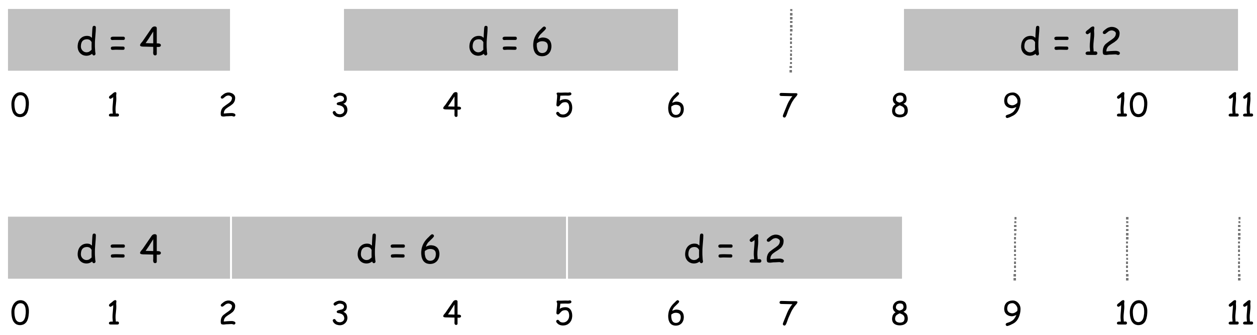
```
Trier les n tâches par date-limite d_1 ≤ d_2 ≤ … ≤ d_n

t ← 0
pour j = 1 à n
   Affecter la tâche j à l'intervalle [t, t + t_j]
   s_j ← t, f_j ← t + t_j
   t ← t + t_j
retourner les intervalles [s_j, f_j]
```



Minimizing Lateness : pas de moment d'oisiveté

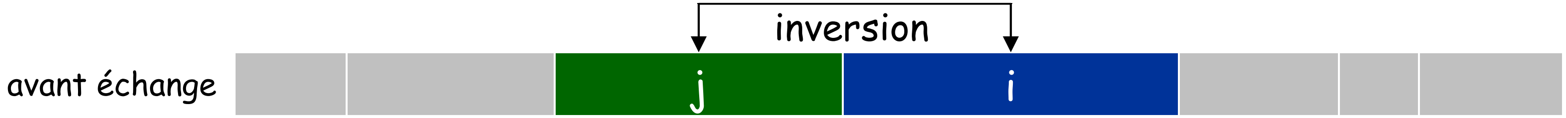
Observation. Il existe une solution optimale sans inactivité (idle time)



Observation. L'approche gloutonne garantit l'absence d'inactivité.

Minimizing Lateness : inversions

Déf. Une inversion idans une programmation S is un couple de tâches (i,j) tel que : i < j mais j est programmé avant i.

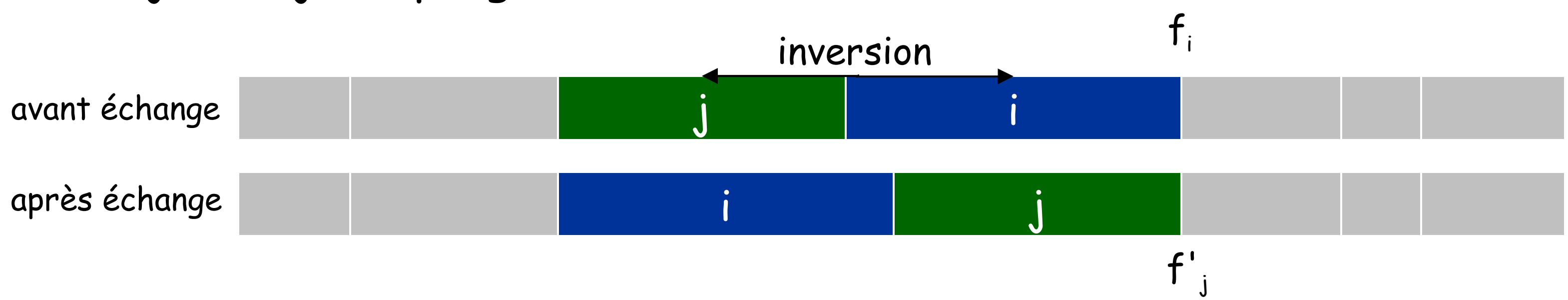


Observation. Dans l'approche gloutonne, il n'y a pas d'inversion.

Observation. Si une programmation (sans inactivité) a une inversion, elle en a une avec un couple inversé programmé de façon contiguë.

Minimizing Lateness: inversions

Déf. Une inversion idans une programmation S is un couple de tâches (i,j) tel que : i < j mais j est programmé avant i.



Remarque. Sl'échange de deux tâches inversées et contiguës réduit le nombre d'inversions d'une unité et n'accroît pas le retard maximal.

Preuve. Soit λ le retard avant l'échange, et λ' le retard après l'échange.

- $\lambda'_k = \lambda_k$ pour tout $k \neq i, j$
- $\lambda'_i \leq \lambda_i$
- Si la tâche j est retardée :

$$
\begin{array}{lll}
\ell'_j & f'_j - d_j & (\text{definition}) \\
 & f_i - d_j & (j \text{ finishes at time } f_i) \\
 & f_i - d_i & (i<j) \\
 & \ell_i & (\text{definition})
\end{array}
$$

Minimizing Lateness: analyse de l'algorithme glouton

Théorème. La programmation gloutonne S est optimale.

Preuve. Soit S* une programmation optimale ayant aussi peu d'inversions que possibles.

- On peut supposer que S* ne programme aucune inactivité.
- Si S* n'a pas d'inversion, alors S = S*.
- Si S* a une inversion, soit i-j une inversion contiguë.
  - échanger i et j n'accroît pas le retard et diminue (strictement) le nombre d'inversions
  - cela contredit la définition de S* ▪

Stratégies d'évaluation de l'approche gloutonne

L'algorithme glouton garde de l'avance. Smontrer qu'après chaque étape de l'algorithme glouton, sa solution (partielle) est au moins aussi bonne que celle de n'importe quel autre algorithme.

L'argument de l'échange. Transformer graduellement une solution optimale en la solution produite par l'algorithme glouton, sans altérer sa qualité.

L'argument structurel. Mettre en évidence un invariant "structurel" simple permettant d'affirmer que toute solution doit avoir au moins une certaine valeur.

Optimal Caching

Optimal Offline Caching

Caching.

- Cache d'une capacité de k items (mots-mémoire).
- Suite de m requêtes $d_1, d_2, \ldots, d_m$.
- Succès (cache hit) : l'item demandé est dans le cache.
- Échec (cache miss) : l'item demandé n'est pas dans le cache : il faut l'y transférer, et remplacer un item présent dans le cache, si celui-ci est plein.

But. Un schéma de remplacement qui minimise les échecs.

Ex: k = 2, cache initial = ab,
requêtes: a, b, c, b, c, a, a, b.

Schéma de remplacement optimal : 2 échecs.

a	a	b
b	a	b
c	c	b
b	c	b
c	c	b
a	a	b
a	a	b
b	a	b
requêtes	cache	

Optimal Offline Caching: Farthest-In-Future

Farthest-in-future (FF). On remplace l'item qui sera demandé le plus tardivement.

cache courant :	a	b	c	d	e	f

requêtes futures : `g a b c e d a b b a c d e a f a d e f g h ...`

échec (sous g) — on remplace celui-ci (sous f)

Théorème. [Bellady, 1960s] FF est un schéma de remplacement.

Preuve. L'algorithme et le théorème sont conformes à l'intuition ; la preuve est subtile.

Schémas de remplacement réduits

Déf. Un schéma de remplacement est reduit s'il ne transfère un item dans le cache qu'au moment où il est demandé.

Intuition. On peut transformer un schéma non réduit en un schéma réduit sans augmenter le nombre d'échecs.

a	a	b	c
a	a	x	c
c	a	d	c
d	a	d	b
a	a	c	b
b	a	x	b
c	a	c	b
a	a	b	c
a	a	b	c

un schéma non réduit

a	a	b	c
a	a	b	c
c	a	b	c
d	a	d	c
a	a	d	c
b	a	d	b
c	a	c	b
a	a	c	b
a	a	c	b

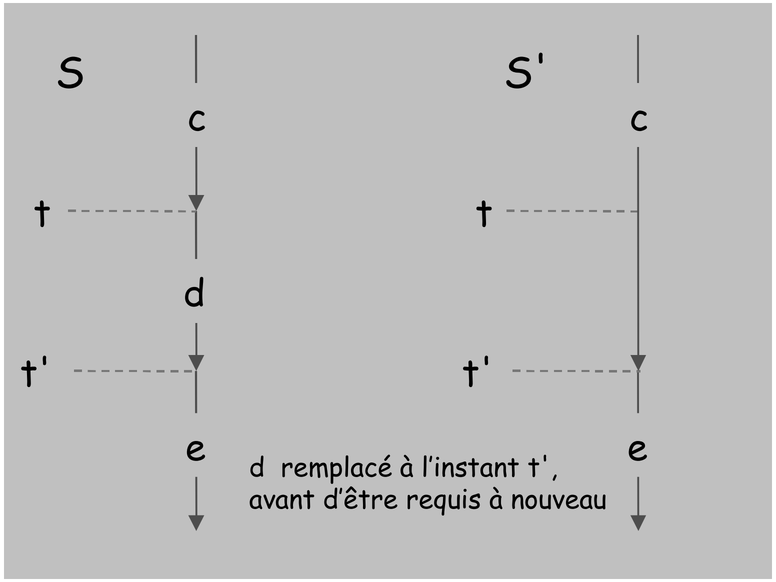
un schéma réduit

Schémas de remplacement réduits

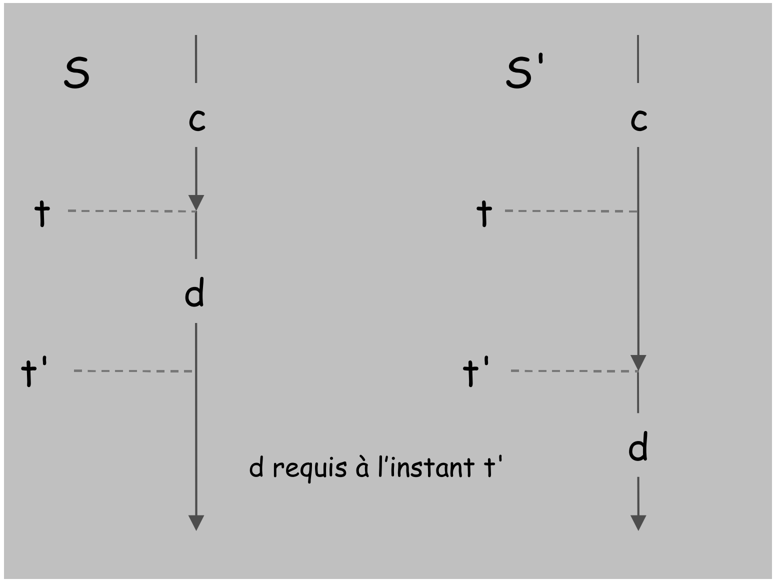
Propriété. Étant donné un schéma de remplacement non réduit S, onpeut le transformer en un schéma réduit S' sans accroître le nombre d'échecs.

Preuve. (par récurrence sur le nombre d'items non réduits)

- Supposons que S transfère d dans le cache à l'instant t, sans requête.
- Soit c l' item que S remplace par d dans le cache.
- Cas 1: d remplacé à l'instant t', avant que d soit encore requis.
- Cas 2: d est requis à l'instant t' avant d'être remplacé. ▪



Cas 1



Cas 2

Farthest-In-Future : analyse

Théorème. FF est un schéma de remplacement optimal.
Preuve. (par récurrence sur le nombre de requêtes j)

Invariant: il existe un schéma réduit optimal S qui fait les mêmes remplacements que S_{FF} pendant le traitement des j+1 premières requêtes.

Soit S le schéma réduit qui vérifie l'invariantau cours des j premières requêtes. On construit S' qui vérifie l'invariant après j+1 requêtes.

- On considère la (j+1)[e] requête $d = d_{j+1}$.
- Puisque S et S_{FF} ont fait les mêmes rempalcement jusque là, le contenu du cache avant le requête j+1 est identique pour S et S'.
- Cas 1 : (d est déjà dans le cache). S' = S satisfont l'invariant.
- Cas 2 : (d n'est pas dans le cache et S et S_{FF} remplacent le même item). S' = S satisfont l'invariant.

Farthest-In-Future : analyse

Preuve. (suite)

- Cas 3 : (d n'est pas dans le cache; S_{FF} remplace e; S remplace f $\neq$ e).
  - On commence la construction de S' à partir de S en remplaçant e plutôt que f

	S			S'		
j	…	e	f	…	e	f
j+1	…	e	d	…	d	f

  - Alors S' et S_{FF} font les mêmes remplacements pendant les j+1 premières requêtes ; on montre que la présence de f dans le cache n'est pas plus coûteuse que la présence de e.

Farthest-In-Future : *analyse*

Soit j' le premier instant après j+1 où S et S' diffère, et soit g l'item requis à l'instant j'.

cette différence doit concerner e ou f (ou les deux)

j'	…	e		…	f
	S			S'	

- Cas 3a : g = e. Cela ne peut pas se produire avec FF car f doit être requis avant e.

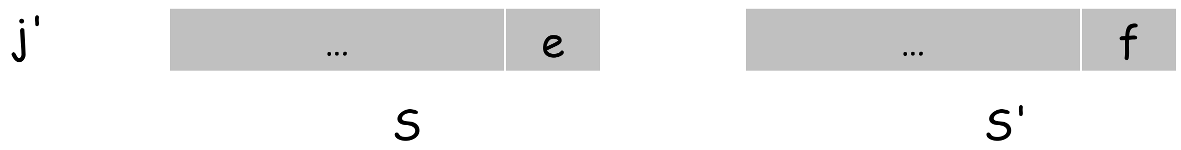
- Cas 3b : g = f. L'élément f ne peut pas être dans le cache de S, on pose donc e' = l'élément que S remplace.
  - si e' = e, S' accède à f dans le cache ; alors S et S' ont le même cache
  - si e' ≠ e, S' remplace e' et transfère e dans le cache; après quoi S et S' ont le même cache

Note : S' n'est plus réduit, mais peut être transformé en un schéma réduit qui "colle" à S_{FF} jusqu'à la requête j+1

Farthest-In-Future : analyse

Soit j' le premier instant après j+1 où S et S' diffère, et soit g l'item requis à l'instant j'.

cette différence doit concerner e ou f (ou les deux)



sinon S' ferait la même action

- Cas 3c : $g \neq e, f$. S doit remplacer e.
  On fait en sorte que S' remplace f ; alors S et S' ont le même cache. ▪

Quelques questions liées au Caching

Algorithmes online vs. algorithmes offline .

- Offline : la suite complète des requêtes est connue a priori.
- Online (plus conforme à la réalité) : les requêtes ne sont pas connues à l'avance.
- Le caching compte parmi les problèmes d'informatique online les plus importants.

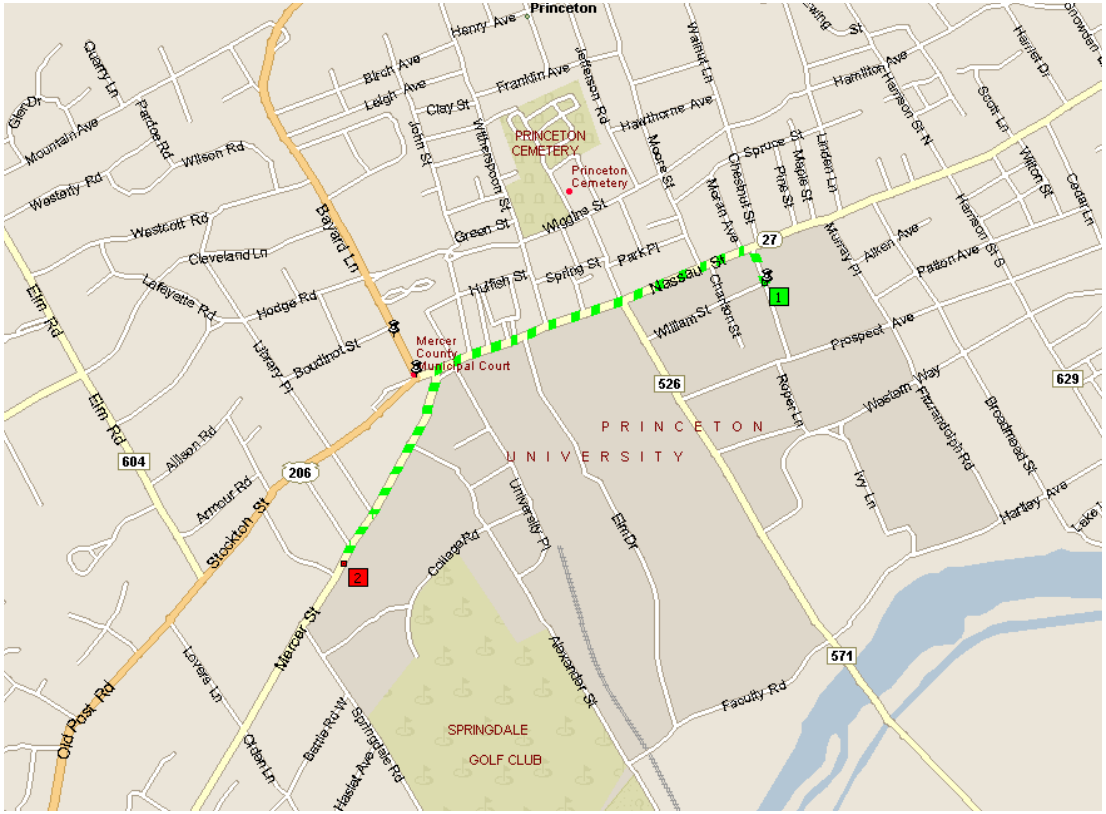
LIFO. Remplace la page transférée le plus récemment.
LRU. Remplace la page requise le moins récemment.

FF avec un renversement de la flèche du temps !

Théorème. FF est un algorithme de remplacement offline optimal.

- C'est une base pour la compréhension et l'analyse des algorithmes online.
- LRU est k-compétitif.
- LIFO est "arbitrairement mauvais".

Plus courts chemins dans un graphe



Plus court chemin du département CS de Princeton à la maison d'Einstein

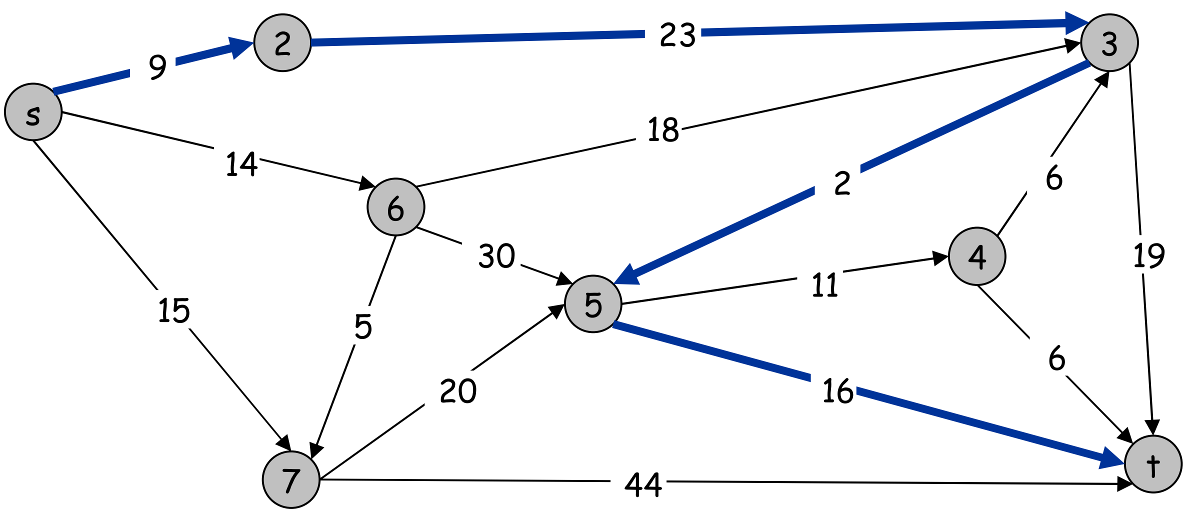
Problème du plus court chemin

Réseau des plus courts chemins.

- Graphe orienté $G = (V, E)$.
- Source s, destination t.
- "coût" (poids) d'un λ_e = longueur de l'arc e. On supposera $\lambda_e \geq 0$

Problème du plus court chemin : trouver un chemin minimal de s à t.

Coût (longueur) du chemin = somme des coûts des arcs du chemin



Coût du chemin s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.

Algorithme de Dijkstra

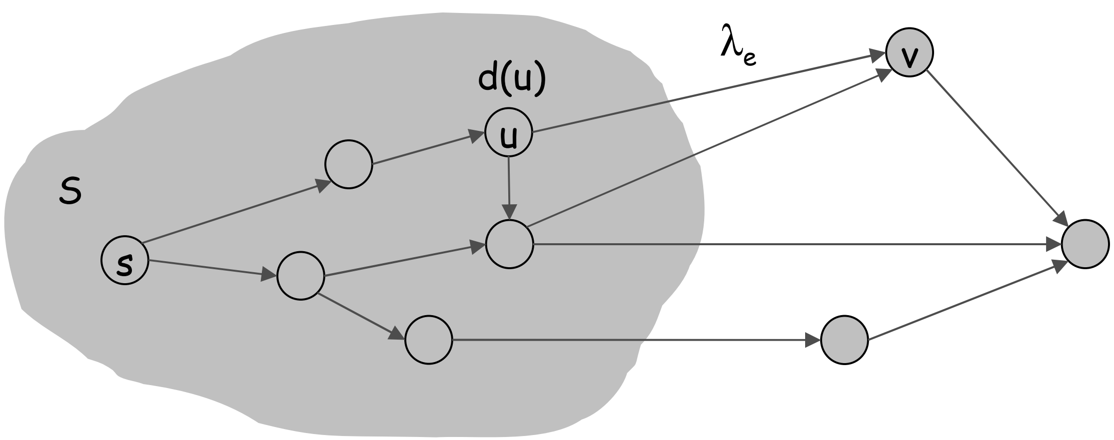
Algorithme de Dijkstra.

- On "maintient" un ensemble de sommets explorés S tel que pour tout $u \in S$, on a déterminé la longueur minimale d(u) d'un chemin de s à u.
- On initialise S = { s }, d(s) = 0.
- À chaque étape, on choisit un sommet v non exploré qui minimise

$$d(u) + \ell_e \, , \text{¿}$$

on ajoute v à S, et on pose d(v) = π(v).

plus court chemin de s à u dans la partie explorée, suivi d'un arc (u, v)



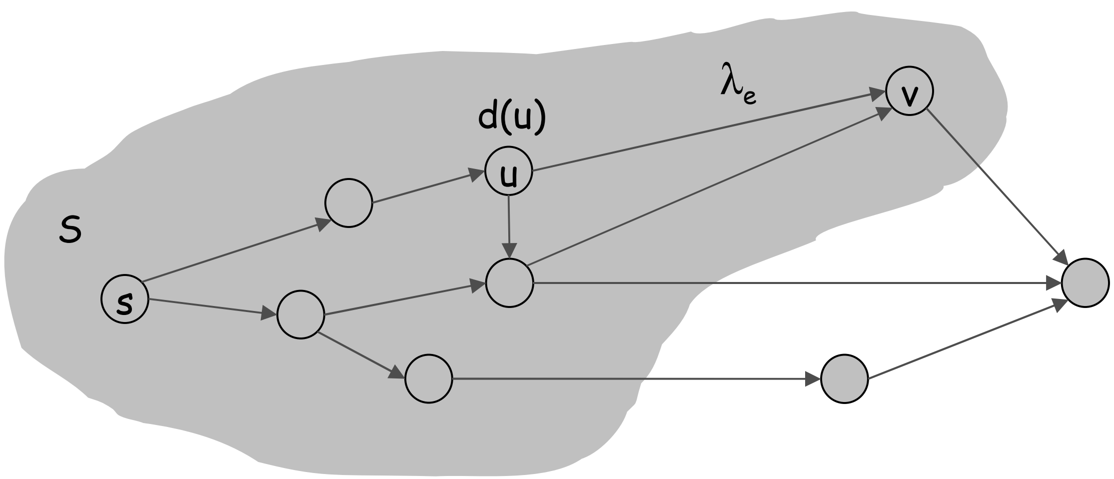
Algorithme de Dijkstra.

- On "maintient" un ensemble de sommets explorés S tel que pour tout $u \in S$, on a déterminé la longueur minimale d(u) d'un chemin de s à u.
- On initialise S = { s }, d(s) = 0.
- À chaque étape, on choisit un sommet v non exploré qui minimise

$$d(u) + \ell_e, ¿$$

plus court chemin de s à u dans la partie explorée, suivi d'un arc (u, v)

on ajoute v à S, et on pose d(v) = π(v).



Algorithme de Dijkstra : correction

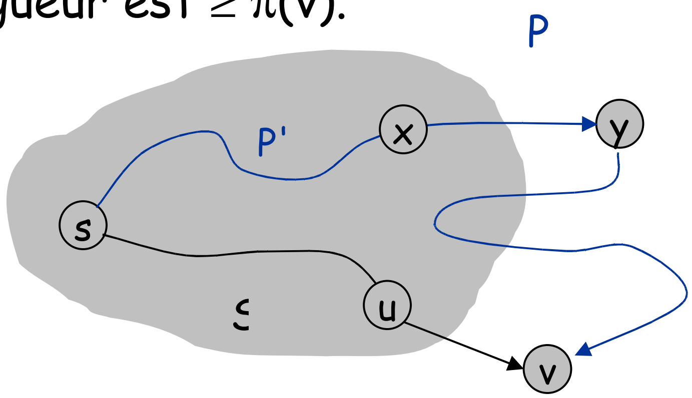
Invariant. Pour chaque sommet $u \in S$, $d(u)$ est la longueur du plus court chemin de s à u.

Preuve. (par récurrence sur $|S|$)

Cas de base : $|S| = 1$ est trivial.

Hypothèse d'induction : on suppose que l'énoncé est vrai pour $|S| = k \geq 1$.

- Soit v le premier sommet ajouté à S, et (u,v) l'arc sélectionné.
- Le chemin minimal s-u suivi de (u, v) est un chemin s-v de longueur $\pi(v)$.
- Soit O un chemin s-v. On montre que sa longueur est $\geq \pi(v)$.
- Soit x-y le premier arc de P à quitter S, et P' le sous-chemin de s à x.
- P est déjà trop long quand il quitte S.



$$\lambda(P) \underset{\substack{\uparrow \\ \text{poids} \geq 0}}{\geq} \lambda(P') + \lambda(x,y) \underset{\substack{\uparrow \\ \text{hypothèse d'induction}}}{\geq} d(x) + \lambda(x,y) \underset{\substack{\uparrow \\ \text{déf de } \pi(y)}}{\geq} \pi(y) \underset{\substack{\uparrow \\ \text{Dijkstra a choisi v et pas y}}}{\geq} \pi(v)$$

Algorithme de Dijkstra : implémentation

Pour chaque sommet non exploré, on maintient explicitement

$$d(u)+l_e \,.\, ¿$$

- Prochain sommet à explorer = sommet avec $\pi(v)$ minimal.
- Pendant l'exploration de w, pour chaque arc e = (v, w), on met à jour

$$p(w)=\min\{p(w),\ p(v)+l_e\}.$$

Implémentation efficace. Maintenir une file de priorité de sommets non explorés PQ, avec $\pi(v)$ comme critère de priorité.

Opération PQ	Dijkstra	Tableau	Tas binaire	Tas d-way	Tas heap
Insertion	n	n	log n	$d \log_d n$	1
Extraction du Min	n	n	log n	$d \log_d n$	log n
Mise à jour des clés	m	1	log n	$\log_d n$	1
Test de vacuité	n	1	1	1	1
Total		n^2	m log n	$m \log_{m/n} n$	m + n log n

Edsger W. Dijkstra

The question of whether computers can think is like the question of whether submarines can swim.

Do only what only you can do.

In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.

The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.

APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.