Concours Centrale-Supelec 2025

Option informatique

Meilleurs itinéraires dans un réseau ferroviaire

I Une file de priorité

I.1 Tas d'appariement

Q. 1 *Écrire une fonction* `min_valeur : 'a arbre -> 'a` *qui renvoie la valeur minimale d'un tas supposé non vide. Quelle est sa complexité dans le pire cas ?*

```
let min_valeur a =
  match a with
  | Vide -> failwith "L'arbre est vide"
  | Noeud (r, fils) -> r
```

La complexité est constante.

Q. 2 *Écrire une fonction fusion :* `'a arbre -> 'a arbre -> 'a arbre` *qui réalise la fusion de deux tas. On gérera les cas où l'un ou les deux tas sont vides.*

```
let fusion a1 a2 =
  match a1, a2 with
  | Vide, _ -> a2
  | _, Vide -> a1
  | Noeud (r1, fils1), Noeud (r2, fils2) ->
      if r1 < = r2
      then Noeud(r1, a2 :: fils1)
      else Noeud (r2, a1 :: fils2)
```

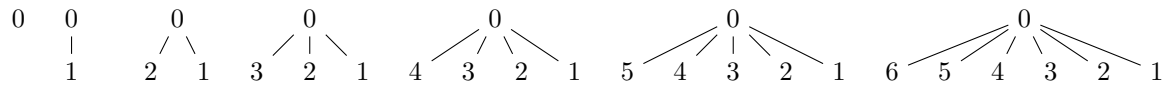
Q. 3 *Écrire une fonction insere : 'a -> 'a arbre -> 'a arbre qui réalise l'insertion d'un élément dans un tas. Quelle est sa complexité dans le pire cas ?*

```
let insere x arbre =
  fusion (Noeud (x, [])) arbre
```

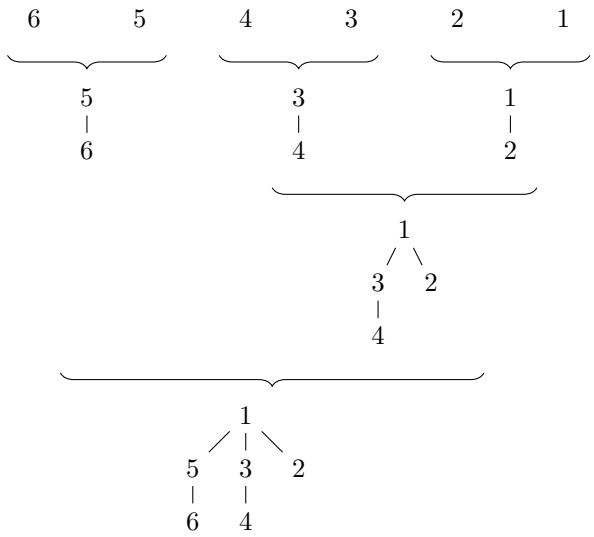
Noter le parenthésage nécessaire de la construction.

Q. 4 *On insère successivement et dans cet ordre les éléments* $0, 1, 2, 3, 4, 5$ *et 6 dans un tas initialement vide, puis on supprime deux fois le minimum. Représenter le tas d'appariement obtenu.*

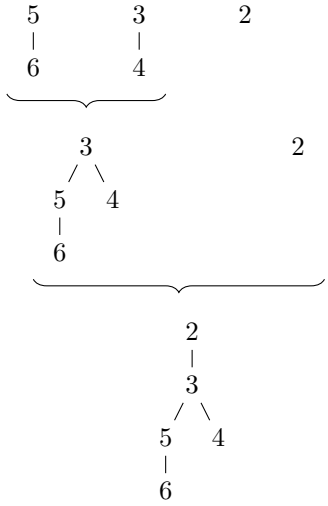
Les insertions donnent



La première suppression du minimum donne



La deuxième suppression du minimum donne



Q. 5 *Écrire une fonction* `fusion_par_paires : 'a arbre list -> 'a arbre` *qui réalise la fusion par paires d'une liste d'arbres. Cette fonction devra renvoyer* `Vide` *pour une liste vide.*

La première étape se fait en accumulant les arbres fusionnés dans une liste. Ils sont alors dans l'ordre inverse, ce qui est l'ordre de fusion de la deuxième étape.

```
let fusion_par_paire liste_a =
  let rec par2 liste_a acc =
    match liste_a with
    | [] -> acc
    | [a] -> a :: acc
    | a1 :: a2 :: reste ->
        par2 reste ((fusion a1 a2) :: acc) in
  let rec recons liste_a =
    match liste_a with
    | [] -> Vide (* dans le cas d'un arbre sans fils *)
    | [a] -> a
    | a1 :: a2 :: reste ->
        recons ((fusion a1 a2) :: reste) in
  recons (par2 liste_a [])
```

Q. 6 *En déduire une* `fonction suppression_min : 'a arbre -> 'a arbre` *qui supprime l'élément minimal d'un tas. Quelle est sa complexité dans le pire cas ?*

```
let suppression_min a =
  match a with
  | Vide -> failwith "Il␣n'y␣a␣rien␣à␣supprimer"
  | Noeud (r, fils) -> fusion_par_paire fils
```

Si p est le nombre de fils du nœud, la première étape de la fusion par paire effectue $\lfloor \frac{p}{2} \rfloor$ fusion et produit une liste de $\lceil \frac{p}{2} \rceil$. La seconde étape effectue alors $\lceil \frac{p}{2} \rceil$ fusions d'où p fusions en tout : comme la fusion s'effectue en temps constant, la complexité est un $\mathcal{O}(p)$.

I.2 Tri par tas

Q. 7 *Écrire une fonction* `tri : 'a list -> 'a list` *permettant de trier une liste dans l'ordre croissant en utilisant un tri par tas d'appariement.*

On commence par une fonction qui construit un arbre à partir d'une liste

```
let construire_arbre elements =
  let rec aux_cons liste arbre =
    match liste with
    | [] -> arbre
    | x :: reste -> aux_cons reste (fusion (Noeud (x, [])) arbre) in
  aux_cons elements Vide
```

On peut ensuite calculer la liste triée

```
let tri liste =
  let rec vider arbre =
    match arbre with
    | Vide -> []
    | Noeud (x, fils) -> x :: (vider (fusion_par_paire  fils)) in
  vider (construire_arbre liste)
```

Q. 8 *Quelle est la complexité dans le meilleur cas de cette fonction tri ? Donner, en justifiant, une forme de liste correspondant à cette complexité.*

La fonction doit construire une liste de taille n donc la complexité doit être au moins linéaire en la taille. Si on part d'une liste triée par ordre décroissant, l'arbre obtenu est une branche unique et les fusions par paire ne doivent traiter que des liste à un seul élément. La complexité est alors optimale en $\mathcal{O}(n)$.

II Optimums de Pareto

Q. 9 *Montrer que les trajets (b) et (f) ne sont pas des optimums de Pareto, en explicitant pour chacun d'entre eux un trajet strictement meilleur.*

$(d) = (5, 1, 40)$ minore strictement $(b) = (5, 1, 50)$ et $(f) = (5, 2, 100)$.

II.1 Notion d'optimum de Pareto et ordre partiel

Q. 10 *Soit X une partie de l'ensemble E telle que X admet un minimum pour l'ordre lexicographique. Montrer que cet élément minimum pour l'ordre lexicographique est un optimum de Pareto sur X, c'est-à-dire un élément minimal de X pour l'ordre produit.*

Soit $x = (x_1, x_2, \ldots, x_d)$ le minimum de X pour l'ordre lexicographique : $x \sqsubseteq y$ pour tout $y \in X$.
Soit $y = (y_1, y_2, \ldots, y_d) \in X$, si on a $x \neq y$, il existe $k \in \{1, 2, \ldots, d\}$ tel que $x_1 = y_1$, $x_2 = y_2$, …, $x_{k-1} = y_{k-1}$ et $x_k < y_k$. Cela rend impossible l'inégalité $y \preceq x$: x est optimum de Pareto sur X.

II.2 Cas de la dimension 1

Q. 11 *Écrire une fonction element_minimal : int list -> int qui renvoie l'élément minimal d'une liste non vide d'entiers.*

```
let rec element_minimalliste =
  match liste with
  | [] -> failwith "La liste ne doit pas être vide"
  | [x] -> x
  | x :: reste -> min t (element_minimal reste)
```

II.3 Cas de la dimension 2

Q. 12 *Donner, sans justification, le ou les éléments minimaux de l'ensemble représenté par* `ex1`.

$(1, 3)$, $(2, 2)$ et $(4, 1)$ sont les éléments minimaux de `ex1`.

Q. 13 *Écrire une fonction* `inferieur : tuple -> tuple -> bool` *telle que* `inferieur x y` *pour deux couples $(x, y) \in \left(\mathbb{Z}^2\right)^2$ s'évalue à* `true` *si $x \preceq y$ et* `false` *sinon, c'est-à-dire si $y \prec x$ ou si x et y ne sont pas comparables.*

```
let inferieur (x1, x2) (y1, y2) =
  (x1 <= y1) && (x2 <= y2)
```

Q. 14 *On considère $X \subseteq \mathbb{Z}^2$ un ensemble qui contient $n \geqslant 1$ couples d'entiers. Donner, en justifiant, un encadrement de card$(\min(X))$, dont les bornes sont atteintes.*

On choisit un élément $x = (x_1, x_2) \in X$ avec $x_1 + x_2$ minimal.
Si on a $y \preceq x$ avec $y = (y_1, y_2)$, alors $y_1 \leqslant x_1$ et $y_2 \leqslant x_2$ donc $y_1 + y_2 \leqslant x_1 + x_2$.
La minimalité de $x_1 + x_2$ implique alors $y_1 + y_2 = x_1 + x_2$ puis
$y_1 \leqslant x_1 = x_1 + x_2 - x_2 = y_1 + y_2 - x_2 \leqslant y_1$ car $y_2 \leqslant x_2$ donc $y_1 = x_1$ et, de même, $y_2 = x_2$.
Ainsi x est un optimum de Pareto donc $\text{card}(\min(X)) \geqslant 1$. Cette borne inférieure est atteinte pour $X = \{(1, 1), (1, 2), \ldots, (1, n)\}$ qui n'admet que $(1, 1)$ comme optimum de Pareto.

On a $\min(X)) \subset X$ donc $\text{card}(\min(X)) \leqslant n$. Cette borne supérieure est atteinte pour $X = \{(1, n), (2, n-1), \ldots, (n, 1)\}$ dans lequel tous les couples sont des optimums (optima ?) de Pareto.

Q. 15 *Donner la trace d'exécution de la fonction* `w` *lors de l'appel à* `elements_minimaux ex1`. *On pourra se contenter de noter uniquement le début de la liste à chaque appel s'il n'y a pas ambiguité sur la valeur de la fin, par exemple ex1 $= [(4, 1); (5, 7); \ldots]$.*

```
-> tri_lexico [(4,1); (5,7); (3,3); (1,3); (2,9); (1,4); (2,2)]
<- [(1,3); (1,4); (2,2); (2,9); (3,3); (4,1); (5,7)]
-> w [(1,3); (1,4); (2,2); (2,9); (3,3); (4,1); (5,7)]
    -> w [(1,3); (2,2); (2,9); (3,3); (4,1); (5,7)]
        -> w [(2,2); (2,9); (3,3); (4,1); (5,7)]
            -> w [(2,2); (3,3); (4,1); (5,7)]
                -> w [(2,2); (4,1); (5,7)]
                    -> w [(4,1); (5,7)]
                        -> w [(4,1)]
                        <- [(4,1)]
                    <- (2,2) :: [(4,1)]
                <- [(2,2); (4,1)]
            <- [(2,2); (4,1)]
        <- (1, 3) :: [(2,2); (4,1)]
    <- [(1, 3); (2,2); (4,1)]
<- [(1, 3); (2,2); (4,1)]
```

Q. 16 *Justifier soigneusement la terminaison de la fonction* `elements_minimaux`.

On a supposé que le tri terminait.
Lors du calcul de `w l` avec `l` liste de longueur $n \geqslant 2$, la fonction fait un appel récursif avec une liste de longueur $n-1$. Pour les listes de taille 0 ou 1 la fonction renvoie un résultat directement donc termine. On prouve donc la terminaison de `w` par récurrence sur la longueur de la liste.
Ainsi la fonction `elements_minimaux` termine.

Q. 17 *Déterminer la complexité de la fonction* `elements_minimaux`.

La fonction commence par un tri, supposé de complexité en $\mathcal{O}\big(n \cdot \log(n)\big)$ pour une liste de longueur n.
Lors de l'appel pour une liste de taille $n \geqslant 2$, la fonction `w` effectue des opérations en temps constant, comparaison, adjonction d'un élément en tête de liste et appel récursif avec une liste de taille $n-1$.
Si on note $C(n)$ la complexité du traitement par `w` d'une liste de taille n, on a ainsi $C(n) \leqslant K + C(n-1)$ pour $n \geqslant 2$ avec $C(1) \leqslant K'$ donc $C(n) = \mathcal{O}(n)$.
La complexité totale est donc en $\mathcal{O}\big(n \cdot \log(n)\big)$.

Q. 18 *Justifier soigneusement la correction de la fonction* `elements_minimaux`.

On numérote les lignes du programme :

```
1 let elements_minimaux ens =
2   let rec w u =
3     match u with
4     | x :: y :: z ->
5       if inferieur x y
6       then w (x :: z)
7       else x :: w (y :: z)
8     | v -> v
9   in w (tri_lexico ens)
```

On va prouver, par récurrence sur la taille de la liste que `w l` renvoie les éléments minimaux de `l` si la liste est triée dans l'ordre lexicographique croissant sans doublon.

Ligne 8 Si la liste est vide, elle n'a pas d'élément minimal et le résultat est la liste vide ; la propriété est prouvée pour les listes de taille 0.
Si la liste n'a qu'un élément, celui-ci est minimal et le résultat est la liste initiale donc contient tous les éléments minimaux ; la propriété est prouvée pour les listes de taille 1.

Ligne 4 Supposons le résultat prouvé pour les listes de taille n avec $n \geqslant 1$.
Soit `l = x :: y :: reste` une liste de taille $n \geqslant 2$ triée pour l'ordre lexicographique sans doublon.
`y :: reste` et `x :: reste` sont encore triées pour l'ordre lexicographique et sans doublon.

Ligne 6 Si $x \preceq y$ alors $x \prec y$ car il n'y a pas de doublon. Ainsi y ne peut pas être un élément minimal donc l'ensemble des éléments minimaux de `l` est l'ensemble des éléments minimaux de `x :: reste`. C'est bien de résultat de `w (x :: reste)` par hypothèse de récurrence.

Ligne 7 Sinon on a $y \prec x$. La question **Q.10** montre que x, minimum pour l'ordre lexicographique est un optimum de Pareto pour `u`. Tous les autres minimums de Pareto pour `u` sont aussi des minimum de Pareto pour `y :: z`, extraite de `u`.

Inversement on doit prouver qu'un minimum de Pareto, y', pour `y :: z` est aussi minimum de Pareto pour `u`. On pose $x = (x_1, x_2)$, $y = (y_1, y_2)$ et $y' = (y'_1, y'_2)$.

On a $x \sqsubset y$ donc $x_1 < y_1$ ou $x_1 = y_1$ et $x_2 < y_2$. Mais on n'a pas $x \prec y$ donc le deuxième cas est impossible d'où $x_1 < y_1$ puis $y_2 < x_2$ car $x_2 \leqslant y_2$ impliquerait $x \prec y$.

On a aussi $y \sqsubset y'$ donc $y_1 \leqslant y'_1$ mais on n'a pas $y \prec y'$ car y' est un minimum de Paréto pour `y :: z` d'où, comme ci-dessus, $y'_2 < y_2$; ainsi $x_1 < y'_1$ et $y'_2 < x_2$ x et y' ne sont pas comparables et y' reste un minimum de Pareto de `u`.

Les minimums de Pareto de `l` sont bien la liste `x :: w (y :: z)`.

II.4 Cas général

Q. 19 *Écrire une fonction* `existe_plus_petit` *de signature* `element -> element list -> bool` *telle que l'appel* `existe_plus_petit u liste` *s'évalue à* `true` *si et seulement s'il existe un élément de la liste inférieur ou égal à l'élément* `u`.

```
let existe_plus_petit u liste =
  match liste with
  | [] -> false
  | v :: reste -> (inferieur v u) || (existe_plus_petit u reste)
```

L'annexe permet d'écrire

```
let existe_plus_petit u = List.exists (fun v -> inferieur v u)
```

Q. 20 *Écrire une fonction* `ajoute_plus_petit` *de signature* `element -> element list -> element list` *telle que* `ajoute_plus_petit u liste` *ajoute l'élément* `u` *à une liste en supprimant de cette liste tous les éléments supérieurs ou égaux à* `u`.

```
let rec ajoute_plus_petit u liste =
  match liste with
  | [] -> [u]
  | v :: reste ->
    let new_liste = ajoute_plus_petit u reste in
    if (inferieur u v) then new_liste else v :: new_liste
```

L'annexe permet d'écrire

```
let ajoute_plus_petit u liste =
  u :: (List.filer (fun v -> not (inferieur u v)) liste)
```

Q. 21 *En déduire une fonction* `elements_minimaux : element list -> element list` *qui renvoie une liste composée des éléments minimaux d'une liste passée en paramètre. On suppose que la liste passée en paramètres ne comporte pas de doublons et représente un ensemble X d'éléments. Cette fonction doit effectuer au plus $\mathcal{O}(nk)$ appels à la fonction* `inferieur`*, où n est le cardinal de X et k la largeur de X, ce que l'on justifiera brièvement.*

```
1 let elements_minimaux liste =
2   let rec etape reste faits =
3     match reste with
4     | [] -> faits
5     | x :: xs ->
6       if existe_plus_petit x faits
7       then etape xs faits
8       else etape xs (ajoute_plus_petit x faits)
9   in etape liste [];;
```

On commence par prouver que, lors de chaque appel de la fonction `etape`, `reste` est un suffixe de la liste, `liste = debut @ reste` et `faits` est l'ensemble des éléments minimaux de `debut` : propriété (†).

- L'appel principal est `etape liste []` qui correspond à `debut = []` pour le quel l'ensemble des éléments minimaux est bien vide.
- Si, à la ligne 5, `faits` est l'ensemble des éléments minimaux de `debut` avec `liste = debut @ reste` et que `reste = x :: xs` est non vide, deux cas sont possibles :
  - soit `x` n'est pas un minimum de `debut @ [x]` ; il existe alors un élément dans `debut` inférieur à `x`, il y a donc un élément minimal de `debut` inférieur à `x`. Cela est testé par la ligne 6 à l'aide le la fonction `existe_plus_petit x faits`. Dans ce cas `faits` est encore l'ensemble des éléments minimaux de `debut @ [x]`, préfixe associé à `xs`, et l'appel `etape xs faits` à la ligne 7 vérifie la propriété (†).
  - soit `x` est un minimum de `debut @ [x]`. Dans ce cas le test de la ligne 6 donnera `false`. les minimums de `debut @ [x]` seront soit `x` soit les minimum de `reste` qui ne sont pas minorés par `x`, ce que construit la fonction `ajoute_plus_petit`. Ici encore l'appel à la fonction `etape` de ligne 8 vérifie la propriété (†).
- Le résultat renvoyé à la ligne 4 correspond à l'ensemble des éléments minimaux du préfixe de la liste vide dans `liste`, c'est-à-dire la liste entière : la fonction renvoie bien la liste des éléments minimaux [1]

Ainsi, à chaque appel de `etape`, `faits` contient des éléments minimums donc non comparables de `liste`, son cardinal est donc majoré par la largeur de la liste k.
On effectue, au total, n appels aux fonctions `ajoute_plus_petit` et `existe_plus_petit` dans lesquelles le nombre d'appels à la fonction `inferieur` est la taille de `faits`, au plus k appels. Il y a donc au plus $2nk$ à la fonction `inferieur`, c'est bien un $\mathcal{O}(nk)$.

1. Le sujet ne demandait pas explicitement de donner la preuve de la fonction mais j'ai besoin de propriété de `faits` pour le calcul de la complexité.

III Éléments minimaux d'un langage régulier

III.1 Éléments minimaux d'un langage

Q. 22 *Montrer que* $\varepsilon \in L \iff \min(L) = \{\varepsilon\}$.

On a $\min(L) \subset L$ donc, si $\min(L) = \{\varepsilon\}$, $\varepsilon \in L$.
Inversement, si $\varepsilon \in L$, alors $\varepsilon \prec u$ pour tout $u \in L \setminus \{\varepsilon\}$ donc ε est un élément minimal de L et aucun autre mot ne peut l'être : $\min(L) = \{\varepsilon\}$.

Q. 23 *Montrer que si* $L \subseteq \Sigma^+$ *est non vide, alors* $\min(L) \neq \emptyset$. *Autrement dit, l'ordre partiel* $\preceq$ *est bien fondé.*

On considère la longueur minimale des mots de L et l'ensemble L_d des mots de L de longueur d. $L - d$ est un ensemble fini car l'alphabet est fini ; il admet donc un élément minimal u_0.
Si u_0 n'était pas minimal dans L, il existerait $u_1 \in L$ tel que $u_1 \prec u_0$.
Par définition on devrait avoir $|u_1| \leqslant |u_0|$ donc $|u_1|$ de longueur minimale t $u_1 \in L_d$, ce qui contredit la minimalité de u_0 dans L_d. On a bien construit un élément minimal de L.

Q. 24 *Donner* $\min(L_1)$ *sans justification puis montrer que les langages* L_1 *et* $\min(L_1)$ *sont réguliers.*

On a $a^ib \prec a^ib^j$ pour tout $j \geqslant 2$. Les seuls éléments minimums possibles sont donc les a^ib.
De plus a^ib et a^jb ne sont pas comparables pour $i \neq j$ donc $\min(L_1) = \left\{a^ib \;\; i \in \mathbb{N}^*\right\}$.
L_1, dénoté par `a*b*`, et $\min(L_1)$, dénoté par `a*b`, sont rationnels.

Q. 25 *(a) Déterminer* $\min(L_2)$*, en expliquant brièvement comment ce langage est obtenu.*
(b) Montrer que les langages L_2 *et* $\min(L_2)$ *ne sont pas réguliers. On détaillera uniquement la preuve montrant que* L_2 *n'est pas régulier, et on justifiera brièvement cette propriété pour* $\min(L_2)$.

(a) Pour $j \geqslant i$ et $k \geqslant \max(i,j)$, $a^ib^jc^k$ et $a^jb^ic^k$ appartiennent à L_2 et $a^jb^ic^k \prec a^ib^jc^k$: les éléments minimums de L_2 sont de la forme $a^ib^jc^k$ avec $j \leqslant i$.
Pour $j \leqslant i$ et $k > i$, $a^ib^jc^k$ et $a^ib^jc^i$ appartiennent à L_2 et $a^ib^jc^i \prec a^ib^jc^k$: les éléments minimums de L_2 sont de la forme $a^ib^jc^i$ avec $j \leqslant i$.
Si on a $u \preceq v = a^ib^jc^i$ avec $j \leqslant i$, on a montré qu'il existe $u' = a^pb^qc^p$ avec $q \leqslant p$ et $u' \preceq u \preceq v$.

- La condition de longueur donne $2p + q \leqslant 2i + j$,
- $u'_{p+1} = b$ impose $v_{p+1} \in \{b, c\}$ donc $i \leqslant p$,
- $u'_{p+q+1} = c$ impose $v_{p+q+1} = c$ donc $i + j \leqslant p + q$.

On a donc $2p + q \leqslant i + j + i \leqslant p + q + i$ donc $p \leqslant i$ puis $p = i$.
$i + j \leqslant p + q$ donne alors $j \leqslant q$ et $2p + q \leqslant 2i + j$ donne $q \leqslant j$ d'où $j = q$.
Ainsi $u' = v$: u n'a pas de minorant autre que lui même.
On en déduit que $\min(L_2) = \left\{a^ib^jc^i \;;\; i, j \in \mathbb{N}^* \wedge j \leqslant i\right\}$.

(b) Si L_2 était régulier on pourrait appliquer le lemme de l'étoile.
Il existerait un entier N tel que tout mot de L_2 de longueur N au moins se décompose en $w = w_1 \cdot w_2 \cdot w_3$ avec w_2 non nul, $|w_1 \cdot w_2| \leqslant N$ et $w_1 \cdot w_2^k \cdot w_3 \in L_2$ pour tout k.
Si on choisit $w = a^Nbc^N$ qui appartient à L_2, la décomposition doit être de la forme $w_1 = a^p$, $w_2 = a^q$ et $w_3 = a^{N-p-q}bc^N$. Or $w_1 \cdot w_2^2 \cdot w_3 = a^{N+q}bc^N$ n'appartient pas à L_2.
La contradiction montre que L_2 n'est pas rationnel.
Comme on a choisi w dans $\min(L_2)$ la même démonstration montre que $\min(L_2)$ n'est pas rationnel.

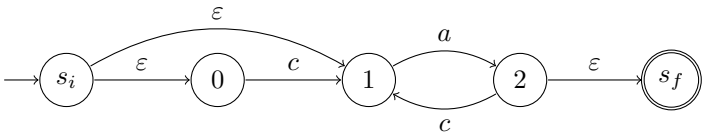
Q. 26 *Montrer que* $\min(L)$ *peut être régulier même si* $L \subseteq \Sigma^+$ *n'est pas régulier.*

On considère $L_3 = \{a^ib^j \;; i > j\}$. L_3 n'est pas régulier car si on pouvait appliquer le lemme de l'étoile avec un entier N, alors $w = a^{n+1}b^n$ qui appartient à L_3, pourrait se décomposer en $w = w_1 \cdot w_2 \cdot w_3$ avec $|w_1 \cdot w_2| \leqslant N$ et $|w_2| > 0$ donc $w_1 = a^p$, $w_2 = a^q$. Pourtant $w_3 = a^{n+1-p-qb^n}$ et $w_1 \cdot w_2^0 \cdot w_3 = a^{N+1-qb^n}$ ne peut appartenir à L_3 car $N + 1 - q \leqslant N$. La contradiction montre que L_3 n'est pas rationnel.
On a $a \in L_3$ et $a \prec w$ pour tout $w \in L_3$ donc $\min(L_3) = \{a\}$ est fini donc rationnel.

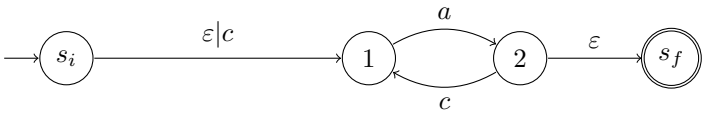
III.2 Automates finis non déterministes

Q. 27 *Appliquer l'algorithme d'élimination des états à l'automate $\mathcal{A}_1$ de la figure 4, en éliminant successivement, dans cet ordre, les états 0,1 puis 2 . En déduire une expression régulière qui dénote le langage $\mathcal{L}(\mathcal{A}_1)$.*

On ajoute les nouveaux états

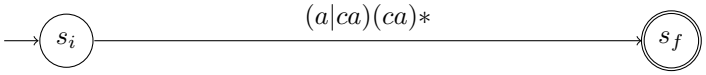


On élimine 0



On élimine 1

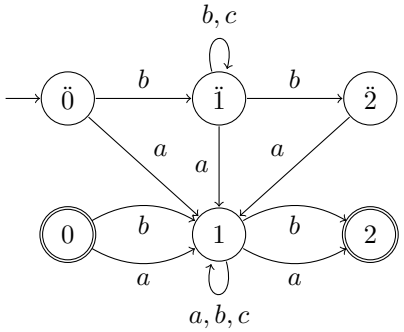
On élimine 2



Ainsi $\mathcal{L}(\mathcal{A}_1)$ est dénoté par `(a|ca)(ca)*`.

III.3 Automates marqués

Q. 28 *Appliquer cette construction sur l'automate marqué $\mathcal{A}_2^M$ de la figure 5 et représenter l'automate ordinaire $\ddot{\mathcal{A}}_2$ ainsi obtenu.*



Q. 29 *Montrer, de manière générale, que $\mathcal{L}(\mathcal{A}^M) = \mathcal{L}(\ddot{\mathcal{A}})$.*

Si $u \in \mathcal{L}(\ddot{\mathcal{A}})$ on considère un calcul réussi pour u.
Le premier état est initial donc appartient à $\ddot{Q}$ est le dernier est final donc appartient à Q. Comme il n'y a pas de transition depuis un état de Q vers un état de $\ddot{Q}$, le calcul peut s'écrire

$$\ddot{q}_0 \xrightarrow{u_1} \ddot{q}_1 \xrightarrow{u_2} \cdots \xrightarrow{u_{i-1}} \ddot{q}_{i-1} \xrightarrow{u_i} q_i \xrightarrow{u_{i+1}} \cdots \xrightarrow{u_{n-1}} q_{n-1} \xrightarrow{u_n} q_n$$

On peut lui associer le calcul réussi dans $\mathcal{A}$: $q_0 \xrightarrow{u_1} q_1 \xrightarrow{u_2} \cdots \xrightarrow{u_{i-1}} q_{i-1} \xrightarrow{u_i} q_i \xrightarrow{u_{i+1}} \cdots \xrightarrow{u_{n-1}} q_{n-1} \xrightarrow{u_n} q_n$ qui contient la transition marquée $q_{i-1} \xrightarrow{u_i} q_i$ donc $u \in \mathcal{L}(\mathcal{A}^M)$.

Inversement, si $u \in \mathcal{L}(\mathcal{A}^M)$, on considère un calcul réussi pour u dans M. Ce calcul contient au moins une transition marquée et on considère la première d'entre elles : $q_{i-1} \xrightarrow{u_i} q_i$.
On obtient un calcul réussi pour u dans $\ddot{\mathcal{A}}$ en remplaçant q_k par $\ddot{q}_k$ pour $0 \leqslant k < i$ d'où $u \in \mathcal{L}(\ddot{\mathcal{A}})$.
On a bien $\mathcal{L}(\mathcal{A}^M) = \mathcal{L}(\ddot{\mathcal{A}})$.

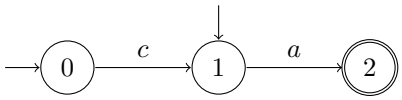
III.4 Automate des minimaux

Q. 30 *Montrer qu'il suffit de montrer que $\breve{L}$ est régulier pour montrer que* $\min(L)$ *est régulier.*

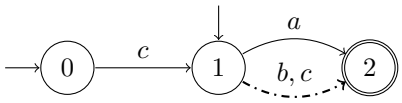
Les éléments minimums de L sont les mots de L qui n'appartient pas à $\breve{L}$.
Ainsi, si $\breve{L}$ est régulier, alors $\min(L) = L \cap \overline{\breve{L}}$ est régulier.

Q. 31 *Appliquer cette construction sur l'automate $\mathcal{A}_1$ de la figure 4, avec l'alphabet Σ_3, et représenter l'automate marqué $\breve{\mathcal{A}}_1^M$ ainsi obtenu.*

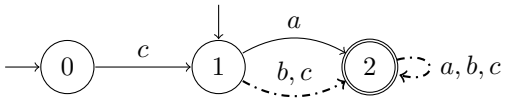
(a)



(b)



(c)



Q. 32 *Montrer, de manière générale, que $\breve{L} \subseteq \mathcal{L}\left(\breve{\mathcal{A}}^M\right)$.*

On considère $u \in \breve{l}$ et $v \in L$ tel que $v \prec u$. On a $|v| \leqslant |u|$, $v_i \leqslant u_i$ pour $1 \leqslant i \leqslant |v|$ et $u \neq v$.

- Si $v_i = u_i$ pour tout $i \in \{1, 2, \ldots, |v|\}$ alors $u = v \cdot w$ avec $w \neq \varepsilon$ car $u \neq v$.
  Un calcul réussi pour v dans $\mathcal{A}$ reste un calcul réussi dans $\breve{\mathcal{A}}$; en lui ajoutant les transitions étiquetées par les lettres de w depuis l'état final atteint vers lui-même on a un calcul réussi pour $u = v \cdot w$. Comme ce calcul finit par des transitions marquées, on a $u \in \mathcal{L}\big(\breve{\mathcal{A}}^M\big)$.
- S'il existe k tel que $v_k < u_k$, on écrit $u = u' \cdot u$ avec $|u'| = |v|$. On transforme un calcul réussi pour v dans $\mathcal{A}$ en un calcul réussi pour u' dans $\breve{\mathcal{A}}$ en remplaçant les transitions étiquetées par v_i par celles étiquetées par u_i. On ajoute ainsi des transitions marquées. On adjoint les transitions de l'éventuel suffixe et on a aussi $u \in \mathcal{L}\big(\breve{\mathcal{A}}^M\big)$.

Dans les deux cas on a $u \in \mathcal{L}\big(\breve{\mathcal{A}}^M\big)$ donc $\breve{L} \subseteq \mathcal{L}\left(\breve{\mathcal{A}}^M\right)$.

Q. 33 *Montrer, de manière générale, que $\mathcal{L}\left(\breve{\mathcal{A}}^M\right) \subseteq \breve{L}$.*

Soit $u \in \mathcal{L}\left(\breve{\mathcal{A}}^M\right)$. On considère un calcul réussi pour u dans $\breve{A}$ contenant au moins une transition marquée :
$q_0 \xrightarrow{u_1} q_1 \xrightarrow{u_2} \cdots \xrightarrow{u_{n-1}} q_{n-1} \xrightarrow{u_n} q_n$. q_k est le premier état final du chemin.
On pose $u' = u_1 u_2 \cdots u_k$: u' est reconnu par $\breve{\mathcal{A}}$.

- Si aucune des transitions $q_{i-1} \xrightarrow{u_i} q_i$ n'est marquée alors u' est reconnu par $\mathcal{A}$ et $u' \neq u$ car il existe une transition marquée. Ainsi $u' \in L$ et $u' \prec u$ donc $u \in \breve{L}$.
- On suppose qu'il existe au moins une transition marquée. Pour chaque transition marquée $q_{i-1} \xrightarrow{u_i} q_i$ il existe transition non marquée $q_{i-1} \xrightarrow{v_i} q_i$ avec $v_i < u_i$. On construit le mot v à partir de u' en remplaçant chaque lettre u_i d'une transition marquée par la lettre v_i correspondante. On obtient un mot v reconnu par $\mathcal{A}$ avec $v \prec u'$. De plus $u' \preceq u$ donc $v \prec u$ d'où $u \in \breve{L}$.

Dans les deux cas on a $u \in \breve{L}$ donc $\mathcal{L}\left(\breve{\mathcal{A}}^M\right) \subseteq \breve{L}$.

Q. 34 *En déduire que si $L \subseteq \Sigma^+$ est un langage régulier alors* $\min(L)$ *est également un langage régulier.*

Si L est régulier donc reconnu par un automate $\mathcal{A}$ alors $\breve{L}$ est reconnu par $\breve{A}^M$ donc est régulier et la question **Q.30** permet d'en déduite que $\min(L)$ est régulier.

IV Meilleurs chemins dans un graphe

Algorithme 1 Algorithme de Dijkstra pour les réseaux ferroviaires

```
1: Initialiser un tableau d qui à chaque sommet associe la liste vide []
2: Soit Q une file de priorité vide
3: Ajouter s) à Q avec la priorité (0, −1, 0)
4: tant que Q est non vide faire
5:     défiler v de Q de priorité π_v minimale pour l'ordre lexicographique
6:     si aucun élément de d[v] n'est plus petit que π_v alors
7:         supprimer de d[v] tous les éléments plus grands que π_v
8:         ajouter π_v dans d[v]
9:         pour tout arc e de v vers w faire
10:            enfiler w dans Q avec la priorité π_v ⊕ (t(e), 1, p(e))
```

Q. 35 *Les files de priorité de la partie I stockent a priori uniquement les priorités. Justifier qu'on peut les utiliser pour implémenter l'algorithme 1 si on stocke dans chaque nœud le couple (π_v, v).*

L'ordre lexicographique sur le couple (π_v, v) correspond à l'ordre lexicographique sur le quadruplet (a, b, c, v) si $\pi_v = (a, b, c)$. La valeur minimale d'un quadruplet sera celle d'un couple de priorité minimale..

Q. 36 *Détailler l'évolution de* `Q` *et* `d` *lors des 3 itérations suivantes de l'algorithme 1.*

On va donner toutes les étapes.

File de priorité `Q`

Étape 0	0 −1 0 0								
Étape 1	1 0 44 1	3 0 6 3	3 0 88 2	4 0 84 4	6 0 60 5				
Étape 2	2 1 76 3	3 0 6 3	3 0 88 2	4 0 84 4	5 1 50 5	6 0 60 5	6 1 120 4		
Étape 3	3 0 6 3	3 0 88 2	4 0 84 4	4 2 110 5	5 1 50 5	6 0 60 5	6 1 120 4		
Étape 4	3 0 88 2	4 0 84 4	4 2 110 5	5 1 40 5	5 1 50 5	6 0 60 5	6 1 120 4		
Étape 5	4 0 84 4	4 1 94 4	4 1 120 5	4 2 110 5	5 1 40 5	5 1 50 5	6 0 60 5	6 1 93 3	6 1 120 4
Étape 6	4 1 94 4	4 1 120 5	4 2 110 5	5 1 40 5	5 1 50 5	5 1 90 5	6 0 60 5	6 1 93 3	6 1 120 4
Étape 7	4 1 120 5	4 2 110 5	5 1 40 5	5 1 50 5	5 1 100 5	6 0 60 5	6 1 93 3	6 1 120 4	

Tableau `d`

	d.(0)	d.(1)	d.(2)	d.(3)	d.(4)	d.(5)
Étape 1	`[(0,-1,0)]`	`[]`	`[]`	`[]`	`[]`	`[]`
Étape 2	`[(0,-1,0)]`	`[(1,0,44)]`	`[]`	`[]`	`[]`	`[]`
Étape 3	`[(0,-1,0)]`	`[(1,0,44)]`	`[]`	`[(2,1,76)]`	`[]`	`[]`
Étape 4	`[(0,-1,0)]`	`[(1,0,44)]`	`[]`	`[(3,0,6); (2,1,76)]`	`[]`	`[]`
Étape 5	`[(0,-1,0)]`	`[(1,0,44)]`	`[(3,0,88)]`	`[(3,0,6); (2,1,76)]`	`[]`	`[]`
Étape 6	`[(0,-1,0)]`	`[(1,0,44)]`	`[(3,0,88)]`	`[(3,0,6); (2,1,76)]`	`[4,0,84]`	`[]`
Étape 7	`[(0,-1,0)]`	`[(1,0,44)]`	`[(3,0,88)]`	`[(3,0,6); (2,1,76)]`	`[4,0,84]`	`[]`

Q. 37 *Écrire une fonction* `dijkstra_pareto : reseau_ferroviaire -> int -> poids list array` *qui prend en paramètres un graphe orienté décrit par ses listes d'adjacence, un sommet d'origine s et qui calcule la liste des poids optimums de Pareto des chemins de s à tous les autres sommets du graphe.*

On doit redéfinir la fonction de comparaison

```
let inferieur (a, b, c) (d, e, f) =
  a <= d && b <= e && c <= f;;
```

Calcul du couple à placer dans la file de priorité en fonction de π_v (ici décomposé en`(a, b, c)`) et du train partant de `s`.

```
let convert (a, b, c) train =
  (a + train.temps, b + 1, c + train.prix), train.dest
```

Ajout d'une liste de couples à la file de priorité

```
let rec inserer_liste a liste =
  match liste with
  | [] -> a
  | x :: xs -> inserer_liste (insere x a) xs
```

On peut transcrire l'algorithme, sous forme récursive car la file est immutable.

```
let dijkstra_pareto reseau s0 =
  let n = Array.length reseau in
  let d = Array.make n [] in
  let rec etape q =
    if q != Vide
    then begin
      let (pi, s) = min_valeur q in
      if not (existe_plus_petit pi d.(s))
      then d.(s) <- ajoute_plus_petit pi d.(s);
      let ajout = List.map (convert pi) reseau.(s) in
      etape (inserer_liste (suppression_min q) ajout)
    end in
  let q0 = insere ((0, -1, 0), 0) Vide  in
  etape q0;
  d
```

Q. 38 *Montrer qu'un chemin non élémentaire d'un sommet u à un sommet v, c'est-à-dire qui passe deux fois par un même sommet, a un poids qui n'est pas un optimum de Pareto parmi les poids des chemins de u à v.*

On considère un chemin non élémentaire $\mathcal{C} = (e_1, \ldots, e_T)$ reliant u à v ; e_k relie s_k à t_k et on suppose que $s_i =$

s_j avec $i < j$. On peut définir un autre chemin de u à v en enlevant la boucle : $\mathcal{C}' = (e_1, \ldots, e_{i-1}, e_j, \ldots, e_T)$.

$$\begin{aligned} w(C) &= \left(\sum_{k=1}^{T} t(e_k), T-1, \sum_{k=1}^{T} p(e_k)\right) \\ &= \left(\sum_{k=1}^{i-1} t(e_k) + \sum_{k=j}^{T} t(e_k), T-1+i-j, \sum_{k=1}^{i-1} p(e_k) + \sum_{k=j}^{T} p(e_k)\right) \oplus \left(\sum_{k=i}^{j-1} t(e_k), j-i, \sum_{k=i}^{j-1} p(e_k)\right) \\ &= w(\mathcal{C}') \oplus \left(\sum_{k=i}^{j-1} t(e_k), j-i, \sum_{k=i}^{j-1} p(e_k)\right) \succ w(\mathcal{C}') \text{ car } \sum_{k=i}^{j-1} t(e_k) \geqslant 0,\ \sum_{k=i}^{j-1} p(e_k) \geqslant 0 \text{ et } j-i > 0 \end{aligned}$$

$\mathcal{C}$ n'est pas un optimum de Pareto.

Q. 39 *Soient s et t deux sommets du graphe et $\mathcal{C}_{s,t}$ un chemin de s à t, dont le poids est un optimum de Pareto parmi tous les chemins Montrer que le poids de tout chemin $\mathcal{C}_{s,u}$, allant de s à u, préfixe de $\mathcal{C}_{s,t}$, est un optimum de Pareto parmi les poids des chemins de s à u.*

Si $\mathcal{C}_{s,t}$ est la concaténation de 2 chemins $\mathcal{C}_{s,u}$ et $\mathcal{C}_{u,t}$ alors $w(\mathcal{C}_{s,t}) = w(\mathcal{C}_{s,u}) \oplus w(\mathcal{C}_{u,t}) \oplus (0,-1,0)$.
Ainsi si $\mathcal{C}_{s,t}$ est un optimum de Pareto et si $\mathcal{C}_{s,u}$ n'était pas un optimum de Pareto alors on pourrait choisir un chemin $\mathcal{C}'_{s,u}$ de s à u tel que $w(\mathcal{C}'_{s,u}) \prec w(\mathcal{C}_{s,u})$. Alors, pour le chemin $\mathcal{C}'_{s,t}$ concaténation des chemins $\mathcal{C}'_{s,u}$ et $\mathcal{C}_{u,t}$, on aurait $w(\mathcal{C}'_{s,t}) \prec w(\mathcal{C}_{s,t})$ ce qui est impossible.
Tous les chemins préfixes d'un optimum de Pareto sont des optimum de Pareto.

Q. 40 *Montrer que, dans l'algorithme 1, si π_v satisfait le test en ligne 6, alors π_v est un optimum de Pareto parmi les poids des chemins de s à v. En déduire une optimisation de l'algorithme 1.*

Lorsque l'on lit un couple (π_v, v) les autres couples dans la file d'attente lui sont supérieurs ou égaux donc les priorités sont aussi supérieures ou égales à π_v pour l'ordre lexicographique. On ajoute éventuellement des couples dans la file d'attente pour lesquels la priorité est $\pi_v \oplus (t, 1, p)$ strictement supérieure à π_v pour l'ordre lexicographique. La priorité de l'élément lu à l'étape suivante est donc supérieure ou égale à π_v.
La suite des priorités lues est croissante.
Si π_v puis π'_v sont lus dans cet ordre depuis la file d'attente et ajoutés à `d[v]`, on ne peut avoir $\pi'_v \prec \pi_v$ car sinon on aurait π'_v strictement inférieur à π_v pour l'ordre lexicographique ce qui contredit la croissance.
Ainsi π_v ne sera jamais supprimé de `d[v]` et on peut optimiser l'algorithme en supprimant la ligne 7.

Remarque Ceci n'est pas la réponse complète à la question. Pour démontrer que les π_v sont des optimums de Pareto, je ne vois pas d'autre moyen que de prouver que tous les optimums de Pareto d'origine `s` sont ajoutés à `d[v]`. C'est l'objet de la question suivante.

Q. 41 *Démontrer la correction de l'algorithme, c'est-à-dire : d'une part qu'à la fin de l'exécution, pour tout sommet v, `d[v]` contient exactement tous les poids des chemins de s à v optimums de Pareto, et d'autre part que l'algorithme termine.*

- Soit $\mathcal{C}_{s,t}$ un chemin, noté aussi $s = u_0 \xrightarrow{e_1} u_1 \xrightarrow{e_1} u_2 \cdots u_{n-1} \xrightarrow{e_n} u_n = t$.
  On a $w\big(\mathcal{C}_{s,t}\big) = (0,-1,0) \oplus \big(t(e_1), 1, p(e_1)\big) \oplus \cdots \oplus \big(t(e_n), 1, p(e_n)\big)$.
  Ainsi, si le chemin est parcouru dans l'algorithme 1, la priorité associée est le poids du chemin.
- Si le poids de $\mathcal{C}_{s,t}$ est un optimum de Pareto, on a vu à la question **Q.39** que $\mathcal{C}_{s,u}$ vérifie la même propriété pour tout préfixe. On note π_i la priorité associée au chemin $\mathcal{C}_{s,u_i}$.
  (π_0, s) est ajouté à la file d'attente à la ligne 3.
  Si (π_i, u_i) est ajouté à la file d'attente alors, lors de sa lecture, π_i est un optimum de Pareto donc est ajouté à `d[ui]` et ses successeurs, en particulier (π_{i+1}, u_{i+1}), sont ajoutés à la file d'attente.
  Par récurrence tous les π_i sont ajoutés dans le tableau `d`, en particulier $w\big(\mathcal{C}_{s,t}\big)$ est ajouté à `d[t]` pour tout chemin optimum de Pareto.
- On a vu à la question précédente que toute priorité, donc tout poids de chemin, ajouté à `d[v]` n'est jamais retiré, il n'admet donc aucun minorant qui est un poids optimum de Pareto car ceux-ci sont ajoutés. C'est donc que toutes les priorités ajoutées à `d[v]` sont des poids optimums de Pareto des chemins de s à v.
- Ainsi `d[v]` contient exactement tous les poids des chemins de s à v optimums de Pareto.

D'après la question la question **Q.38**, un chemin non élémentaire n'est pas un optimum de Pareto, son poids ne peut pas être ajouté dans le tableau `d`. Dans ce cas ses successeurs ne sont pas traités.
Comme tout chemin de longueur supérieure ou égale à $|V|$ ne peut être élémentaire, seuls des chemins de longueur au plus $|V|$ sont introduits dans la file d'attente. L'algorithme termine donc car il ne traite qu'un nombre fini de cas.

Q. 42 *Proposer une fonction en OCaml permettant d'obtenir la liste de tous les chemins entre un sommet s et un sommet t dont le poids est un optimum de Pareto. On indiquera la signature de la fonction ainsi qu'une rapide description de la structure utilisée.*

Pour déterminer les chemins optimums, on peut commencer par créer le graphe dont les sommets sont les couples priorité/sommet lorsque la priorité réalise un optimum de Pareto pour le sommet, cela se voit dès qu'une priorité n'est pas minorée dans le tableau des optimums, d'après la question **Q.40**. Les arêtes joignent deux optimums séparées par un train.
Pour cela on place dans la file d'attente des couples de couples de priorité/sommet.
Il faut gérer le cas du sommet initial pour ne pas ajouter une boucle entre le sommet initial et lui-même.
De plus il peut exister deux chemins différents qui parviennent à la même priorité ; il faut alors ajouter une arête sans ajouter d'optimum dans le tableau.
Le graphe est codé à l'aide d'une table de hachage dont les clés sont des couples priorité/sommet et les valeurs sont des listes de tels couples.
On commence par l'ajout d'une arête.

```
open Hashtbl

let add_arete g s t =
  if not (mem g t)
  then add g t [s]
  else replace g t (s :: (find g t))
```

On code le graphe transposé pour chercher les chemins à partir du sommet d'arrivée.

```
1  let graphe_dijkstra_pareto reseau s0 =
2    let n = Array.length reseau in
3    let d = Array.make n [] in
4    let g = create n in
5    add g ((0, -1, 0), s0) [];
6    let rec etape q =
7      if q != Vide
8      then begin
9        let ((pi, s), prev) = min_valeur q in
10       if not (existe_plus_petit pi d.(s))
11       then begin
12         d.(s) <-  pi :: d.(s);
13         if s <> s0 then add_arete g prev (pi, s);
14         let modifiee = List.map (convert pi) reseau.(s) in
15         let ajout = List.map (fun x -> (x, (pi, s))) modifiee in
16         etape (inserer_liste (suppression_min q) ajout)
17       end
18       else begin
19         if List.mem pi d.(s)
20         then add_arete g prev (pi, s);
21         etape (suppression_min q)
22       end
23     end in
24   let q0 = insere (((0, -1, 0), s0), ((0, 0, 0), s0)) Vide in
25   etape q0;
26   d, g
```

Voici les différences avec le programme de la question **Q.37**

Ligne 5 On ajoute le sommet correspondant au sommet initial.

Ligne 10 Traitement du cas d'un nouveau sommet atteint

Ligne 12 On applique l'optimisation de la question **Q.40**

Ligne 15 On ajoute l'origine du nouveau couple priorité/sommet

Ligne 19-20 On ajoute une nouvelle arête si un sommet déjà atteint est atteint de nouveau

On peut alors lires chemins depuis leur but

```
let chemins_vers reseau s0 t =
  let d, g = graphe_dijkstra_pareto reseau s0 in
  let rec chemins_vers (pi, s) =
    if s = 0
    then [[]]
    else let avant = List.map chemins_vers (find g (pi, s)) in
      let tout = List.flatten avant in
      let completer = List.map (fun x -> (pi, s) :: x) in
      completer tout in
  let dest = List.map (fun x -> (x, t)) d.(t) in
  List.map List.rev (List.flatten (List.map chemins_vers dest));;
```