CONCOURS CENTRALE•SUPÉLEC

Option informatique

MP

4 heures — Calculatrice autorisée

2025

Meilleurs itinéraires dans un réseau ferroviaire

La recherche de l'itinéraire le plus court entre deux points d'un graphe pondéré est un problème facilement résolu par plusieurs algorithmes. Cette recherche est plus difficile dans un réseau ferroviaire, où les trajets sont restreints par des horaires de départ et d'arrivée à chaque arrêt, et du fait de l'existence de correspondances.

De plus, la notion de « meilleur trajet » se fait souvent en considérant non pas uniquement la durée totale du trajet, mais également d'autres critères tels que le prix du billet, le nombre de correspondances, l'heure de départ, les services à bord, etc. Il est rare d'obtenir un trajet qui minimise simultanément tous les critères : la minimisation d'un critère particulier se faisant souvent au détriment d'un autre. Certains trajets sont cependant moins bons sur tous les critères : ceux-ci n'ont pas besoin d'être considérés.

Ce sujet comporte quatre parties :

- la partie I étudie l'implémentation d'une file de priorité et d'un tri par tas ;
- la partie II définit et étudie la notion d'optimum de Pareto. Les notations introduites dans cette partie sont utilisées dans les parties suivantes ;
- la partie III étudie le calcul d'optimums de Pareto dans le cas d'un langage sur un alphabet fini ;
- la partie IV étudie le calcul d'optimums de Pareto dans un graphe pondéré.

Les parties III et IV sont indépendantes.

On trouvera dans l'annexe A en page 10 quelques points de rappels de syntaxe et de fonctions OCaml.

Consignes aux candidates et candidats Il doit être répondu aux questions de programmation en utilisant le langage OCaml. En cas d'écriture d'une fonction non demandée par l'enoncé, il doit être précisé son rôle, ainsi que sa signature (son type). Lorsque cela est pertinent, la description du fonctionnement des programmes qui ont été écrits est également incitée. On autorise toutes les fonctions des modules `Array` et `List`, ainsi que les fonctions de la bibliothèque standard (celles qui s'écrivent sans nom de module, comme `max`, `failwith` ainsi que les opérateurs comme `@`). Sauf précision de l'énoncé, l'utilisation d'autres modules est interdite.

Lorsqu'une question de programmation demande l'écriture d'une fonction, la signature de la fonction demandée est indiquée dans la question. Les candidates et candidats peuvent écrire une fonction dont la signature est compatible avec celle demandée. Par exemple, si l'énoncé demande l'écriture d'une fonction `int list -> int` qui renvoie le premier élément d'une liste d'entiers, écrire une fonction `'a list -> 'a` qui renvoie le premier élément d'une liste d'éléments quelconques sera considéré comme correct.

Il est possible d'admettre le résultat d'une question, y compris de supposer qu'une fonction demandée a été écrite, afin de traiter les questions suivantes.

I – Une file de priorité

L'objectif de cette première partie est d'implémenter une file de priorité persistante en utilisant une structure de tas d'appariement. Cette structure de tas permet également d'obtenir un tri de complexité optimale.

On peut voir les tas d'appariement comme des arbres n-aires auto-ajustés, c'est-à-dire pour lesquels les opérations sur la structure de données réorganisent l'arbre tout en réalisant globalement une forme d'ajustement. Ils disposent d'une implémentation très simple, tout en étant efficace en pratique, par exemple pour une utilisation comme file de priorité dans l'algorithme de Dijkstra, qui sera étudié dans la partie IV.

Contrairement à une implémentation classique en utilisant un tas binaire implémenté dans un tableau, les opérations sur un tas d'appariement à n éléments pourront avoir un coût en $\mathcal{O}(n)$ dans le pire cas. Cependant, on peut montrer qu'une séquence de n opérations consécutives sur cette structure de données a un coût total en $\mathcal{O}(n \log n)$, ce qui rend l'utilisation de cette structure tout aussi efficace lorsque l'on effectue une séquence d'opérations.

Dans toute cette partie on utilise la relation d'ordre totale implicite en OCaml sur les objets de type `'a` donnée par les opérateurs `<` ou `<=` ou encore par les fonctions `min` et `max`. En particulier, si les objets considérés sont des tuples, alors cet ordre correspond à l'ordre lexicographique sur les composantes de ces tuples.

I.1 – Tas d'appariement

Pour implémenter un tas d'appariement, on utilise des arbres n-aires étiquetés, dont les nœuds possèdent un nombre quelconque d'enfants. La figure 1 présente un exemple de tas d'appariement comportant 12 nœuds.

Dans ce sujet, un *arbre* est soit un arbre vide, soit un nœud formé d'une étiquette et d'une liste éventuellement vide d'arbres *non vides*, qui sont les *enfants* de ce nœud *parent*. Une *feuille* est un nœud qui ne possède pas d'enfants, c'est-à-dire dont la liste d'enfants est la liste vide. On représente en OCaml un arbre à l'aide du type suivant :

```
type 'a arbre =
  | Vide
  | Noeud of 'a * 'a arbre list
```

On remarquera que le constructeur `Vide` sert *uniquement* à dénoter l'arbre vide et ne peut pas apparaître dans une liste d'enfants puisque les enfants d'un nœud sont systématiquement supposés non vides.

Un *tas d'appariement* est un arbre vérifiant la propriété de *tas minimum* : l'étiquette d'un nœud est inférieure ou égale à celle de tous ses enfants.

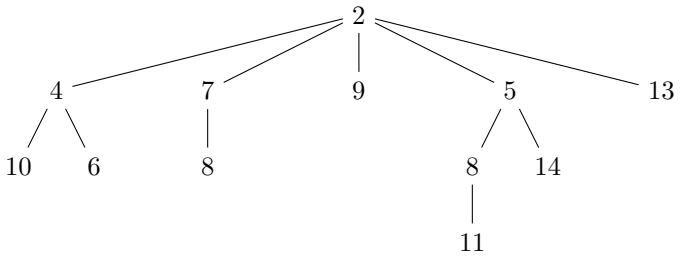

Figure 1 – Un tas d'appariement à 12 nœuds.

Q1. 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 ?

Pour fusionner deux tas non vides, il suffit d'ajouter le tas dont l'étiquette de la racine est la plus grande comme nouvel enfant de l'arbre dont l'étiquette de la racine est la plus petite. Un nouvel enfant est ajouté en tête de la liste des enfants, donc devient le premier enfant à gauche. La complexité de cette opération est donc en $\mathcal{O}(1)$ dans le pire cas. La figure 2 illustre un certain nombre de fusions : les deux arbres au-dessus d'une accolade sont fusionnés pour obtenir l'arbre sous celle-ci.

Q2. Écrire une fonction `fusion : 'a arbre -> 'a arbre -> 'a arbre` qui réalise la fusion de deux tas. On gèrera les cas où l'un ou les deux tas sont vides.

Pour ajouter un élément à un tas, on crée un nouveau tas contenant cet élément et on le fusionne avec le tas initial.

Q3. 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 ?

Pour supprimer l'élément minimal d'un tas, il suffit de supprimer la racine et de fusionner les enfants de celle-ci. Pour obtenir une forme d'auto-ajustement, on procède en deux étapes pour fusionner une liste de k arbres :

(a) on fusionne, *de gauche à droite*, les arbres deux par deux, tant que c'est possible ;

(b) on fusionne progressivement, en un seul arbre, les $\lceil k/2 \rceil$ arbres ainsi obtenus en prenant les arbres un par un, *de la droite vers la gauche*, en partant de l'arbre de droite.

Cette opération en deux étapes est appelée *fusion par paires* et donne le nom à cette structure de *tas d'appariement*.

La figure 2 illustre la suppression de la racine d'étiquette 2 du tas de la figure 1. Lors de la première étape de la fusion par paires, on effectue la fusion des enfants de gauche à droite deux par deux. Les arbres de racine 4 et 7 sont fusionnés entre eux, puis les arbres de racines 9 et 5 sont fusionnés entre eux. L'arbre de racine 13 étant seul, il reste inchangé. Lors de la deuxième étape de la fusion par paires, on fusionne les arbres ainsi obtenus de droite à gauche. Les arbres de racine 5 et 13 sont fusionnés, puis le résultat de cette fusion est fusionné avec l'arbre de racine 4.

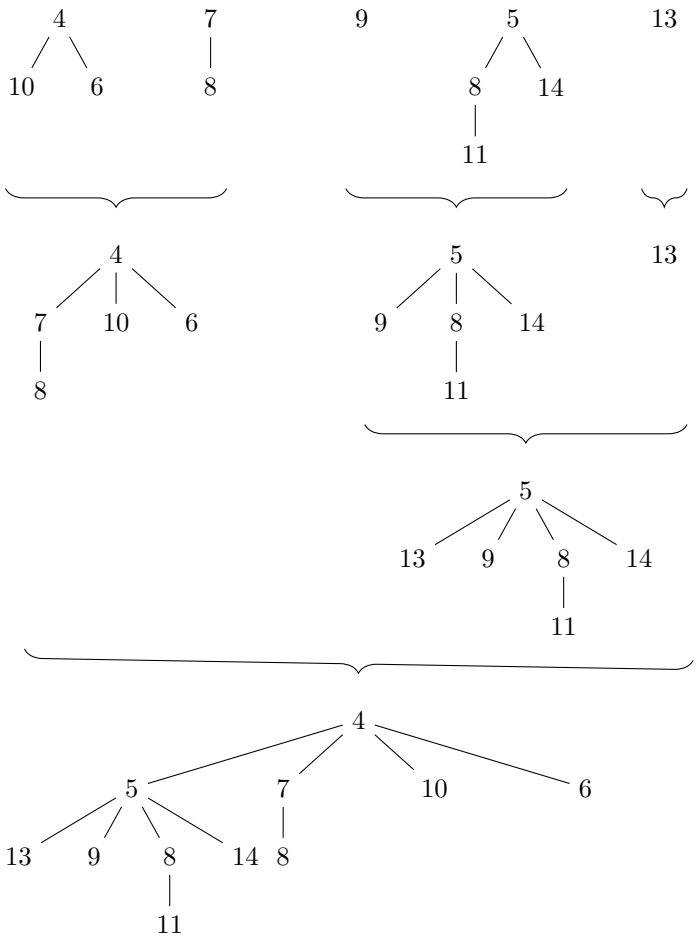

Figure 2 – Fusion par paires des enfants de la racine d'étiquette 2 du tas de la figure 1.

Q4. 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.

Q5. 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.

Q6. 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 ?

On admet qu'une séquence quelconque de n opérations `min_valeur`, `insere` et `suppression_min` sur cette structure de données, à partir d'un tas initialement vide, a une complexité totale en $\mathcal{O}(n \log n)$.

I.2 – Tri par tas

On peut utiliser les fonctions précédentes pour implémenter un tri par tas de complexité $\mathcal{O}(n \log n)$ pour trier une liste de n éléments.

Q7. 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.

Q8. Quelle est la complexité dans le meilleur cas de cette fonction `tri` ? Donner, en justifiant, une forme de liste correspondant à cette complexité.

II – Optimums de Pareto

Dans un problème d'optimisation classique, on cherche généralement à minimiser une certaine quantité parmi un ensemble de solutions possibles. Par exemple, on peut vouloir déterminer le trajet le plus court parmi tous les trajets possibles entre deux points. Lorsque l'on dispose de plusieurs objectifs à minimiser simultanément, ce problème devient cependant bien plus délicat : certains objectifs peuvent être contradictoires et il peut être impossible de les minimiser tous simultanément.

Considérons l'exemple de la figure 3, correspondant à la recherche d'un billet de train entre la gare de Lyon Part Dieu et la gare de Tours sur le réseau ferroviaire national. Pour chaque trajet possible, on retient dans cet exemple trois critères : sa durée, le nombre de correspondances (c'est-à-dire le nombre de fois où l'on doit descendre d'un train pour continuer le trajet avec un autre) et son prix. Idéalement, on souhaite obtenir à la fois un trajet dont la durée est minimale, qui comporte le moins de correspondances possibles et qui est le moins cher.

On observe sur la figure 3 qu'il n'est pas possible de minimiser simultanément tous ces critères. En revanche, il est clairement inutile de considérer le trajet (e) qui a la même durée et le même nombre de correspondances que le trajet (d), mais qui est strictement plus cher que celui-ci.

identifiant	durée (h)	nombre de correspondances	prix (€)
(a)	6	0	60
(b)	5	1	50
(c)	4	2	110
(d)	5	1	40
(e)	5	1	90
(f)	5	2	100
(g)	4	1	120

Figure 3 – Une liste de trajets proposés par un moteur de recherche d'itinéraires pour aller d'une gare à une autre.

On ne sait pas *a priori* quels seront les critères privilégiés par le voyageur. L'objectif est alors de lui présenter l'ensemble des solutions minimales, au sens où aucune autre solution n'est strictement meilleure : il s'agit des *optimums de Pareto*. Par exemple, dans la figure 3 on retiendrait les quatre trajets (a), (c), (d) et (g), qui sont ici les quatre optimums de Pareto. Remarquons que le trajet (a) minimise le nombre de correspondances, que le trajet (d) minimise le prix et que les deux trajets (c) et (g) minimisent la durée, sans pour autant qu'aucun des deux ne soit meilleur que l'autre sur les deux autres attributs.

Q9. 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.

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

On rappelle qu'une *relation d'ordre* $\preceq$ sur un ensemble E est une relation binaire réflexive ($\forall x \in E, x \preceq x$), antisymétrique ($\forall (x,y) \in E^2, x \preceq y \wedge y \preceq x \Rightarrow x = y$) et transitive ($\forall (x,y,z) \in E^3, x \preceq y \wedge y \preceq z \Rightarrow x \preceq z$). Une relation d'ordre est *totale* si tous les éléments sont deux à deux comparables, c'est-à-dire si $\forall (x,y) \in E^2, x \preceq y \vee y \preceq x$. On dit que l'ordre est *partiel* sinon. On note $\prec$ la relation d'ordre stricte associée, définie par : $x \prec y$ si $x \preceq y \wedge x \neq y$.

Si $X \subseteq E$ est une partie de E, un *élément minimal* de X est un élément $x \in X$ qui n'est strictement supérieur à aucun autre élément de X. On note $\min(X)$ l'ensemble des éléments minimaux de X, c'est-à-dire :

$$\min(X) = \{x \in X \mid \nexists y \in X, y \prec x\} = \{x \in X \mid \forall y \in X, y \preceq x \Rightarrow y = x\}.$$

Notons que si la relation d'ordre $\prec$ n'est pas totale, il peut y avoir plusieurs éléments minimaux, qui sont alors deux à deux non comparables.

Soient $d \in \mathbb{N}^*$ et d ensembles totalement ordonnés $(A_1, \leq_1), (A_2, \leq_2), \ldots, (A_d, \leq_d)$, appelés attributs. Pour $i \in [\![1, d]\!]$, $\leq_i$ est donc un ordre total sur A_i. On peut munir $E = A_1 \times A_2 \times \cdots \times A_d$ de différentes relations d'ordre construites à partir des ordres $\leq_i$.

Ordre lexicographique On définit l'ordre lexicographique $\sqsubseteq$ sur E par $(x_1, x_2, \ldots, x_d) \sqsubseteq (x'_1, x'_2, \ldots, x'_d)$ si et seulement si $(x_1, x_2, \ldots, x_d) = (x'_1, x'_2, \ldots, x'_d)$ ou bien, en notant $k = \min\{i \in [\![1, d]\!] | x_i \neq x'_i\}$, $x_k <_k x'_k$. L'ordre lexicographique est un ordre total sur E.

Ordre produit On définit l'ordre produit $\preceq$ par $(x_1, x_2, \ldots, x_d) \preceq (x'_1, x'_2, \ldots, x'_d)$ si $\forall i \in [\![1, d]\!], x_i \leq_i x'_i$. En général, l'ordre produit est seulement un ordre partiel sur E. Un optimum de Pareto est un élément minimal pour cet ordre.

Q10. 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.

Pour simplifier, on suppose dans toute la suite de cette partie que $\forall i \in [\![1, d]\!]$, $(A_i, \leq_i) = (\mathbb{Z}, \leq)$ pour l'ordre total $\leq$ usuel sur les entiers. On s'intéresse donc, dans cette partie, à l'ordre partiel produit sur $\mathbb{Z}^d$.

II.2 – Cas de la dimension 1

Dans le cas de la dimension $d = 1$, l'ordre produit coïncide avec l'ordre total $\leq$ sur $\mathbb{Z}$. Toute partie finie non vide de $\mathbb{Z}$ admet exactement un optimal de Pareto qui est le plus petit élément de cette partie.

Q11. Écrire une fonction `element_minimal : int list -> int` qui renvoie l'élément minimal d'une liste non vide d'entiers.

II.3 – Cas de la dimension 2

Dans cette partie, on s'intéresse à des éléments de $\mathbb{Z}^2$ ordonnés avec l'ordre produit $\preceq$. On manipule donc des objets du type :

```
type tuple = int * int
```

On représente un ensemble de couples par une liste sans doublons. On considère l'exemple suivant :

```
let ex1 = [(4, 1); (5, 7); (3, 3); (1, 3); (2, 9); (1, 4); (2, 2)]
```

Q12. Donner, sans justification, le ou les éléments minimaux de l'ensemble représenté par la liste de couples `ex1`.

Q13. Écrire une fonction `inferieur : tuple -> tuple -> bool` telle que `inferieur x y` pour deux couples $(x, y) \in (\mathbb{Z}^2)^2$ 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.

Q14. On considère $X \subseteq \mathbb{Z}^2$ un ensemble qui contient $n \geq 1$ couples d'entiers. Donner, en justifiant, un encadrement de $\text{card}(\min(X))$, dont les bornes sont atteintes.

On suppose disposer d'une fonction de tri de signature `tri_lexico : tuple list -> tuple list` qui permet de trier une liste d'éléments dans l'ordre croissant pour l'ordre lexicographique. On suppose que cette fonction termine toujours et que la complexité de cette fonction de tri est en $\mathcal{O}(n \log n)$ sur une liste de taille n. La fonction réalisée dans la partie I convient à cet effet.

On considère la fonction suivante, de type `tuple list -> tuple list`, associée à la spécification : si `ens` est une liste de couples représentant un ensemble X d'éléments de $\mathbb{Z}^2$, l'appel `elements_minimaux ens` s'évalue en une liste contenant exactement les éléments minimaux de X.

```
let elements_minimaux ens =
  let rec w u =
    match u with
    | x :: y :: z ->
      if inferieur x y then w (x :: z)
      else x :: w (y :: z)
    | v -> v
  in w (tri_lexico ens)
```

Les noms de variables de cette fonction ont été volontairement rendus inintelligibles.

Pour représenter la trace d'exécution d'un appel à une fonction, on note ligne par ligne dans l'ordre chronologique d'exécution : `-> f args` pour indiquer qu'on commence un appel à `f` sur les paramètres `args` ; ou `<- valeur de retour` pour indiquer la fin de l'appel et la valeur renvoyée. On indente les appels imbriqués. Voici un exemple de fonction et la trace associée à l'appel `cardinal [32; 52]` :

```
let rec cardinal l =
  match l with
  | [] -> 0
  | _ :: tl -> 1 + cardinal tl
```

```
-> cardinal [32; 52]
  -> cardinal [52]
    -> cardinal []
    <- 0
  <- 1
<- 2
```

Q15. 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 ambiguïté sur la valeur de la fin, par exemple `ex1 = [(4, 1); (5, 7) ;...]`.

Q16. Justifier soigneusement la terminaison de la fonction `elements_minimaux`.

Q17. Déterminer la complexité de la fonction `elements_minimaux`.

Q18. Justifier soigneusement la correction de la fonction `elements_minimaux`.

II.4 – Cas général

On se place désormais dans le cadre de la dimension $d \in \mathbb{N}^*$ et même, plus généralement, dans le cadre d'un ordre partiel quelconque sur un ensemble. On suppose ainsi uniquement disposer d'un type abstrait `element` et d'une fonction `inferieur : element -> element -> bool` qui détermine si un élément est inférieur ou égal à un autre pour cet ordre partiel.

Si $(E, \preceq)$ est un ensemble partiellement ordonné et si $X \subseteq E$, la *largeur* de X est le nombre maximal d'éléments simultanément deux à deux non comparables de X ; c'est-à-dire le cardinal maximal d'une partie de X composée uniquement d'éléments deux à deux non comparables :

$$\text{largeur}(X) = \max\{\text{card}(Y) \mid Y \subseteq X \text{ et } \forall (x,y) \in Y^2,\ x \neq y \Rightarrow x \not\preceq y \wedge y \not\preceq x\}.$$

Q19. 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`.

Q20. 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`.

Q21. 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.

III – Éléments minimaux d'un langage régulier

Dans cette partie, on se propose de généraliser la notion d'optimum de Pareto dans le cas d'un nombre variable d'attributs. On pourra ainsi être amené à comparer des tuples de longueurs différentes. À cet effet, on modélise un tuple par un mot sur un alphabet fini.

III.1 – Éléments minimaux d'un langage

On considère un alphabet Σ fini non vide muni d'un ordre total $\leq$ sur les lettres. On rappelle que Σ^* désigne l'ensemble de tous les mots sur Σ et que $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ désigne l'ensemble des mots de longueur non nulle sur Σ.

On considère la relation d'ordre $\preceq$ sur Σ^* définie par : $u_1u_2\dots u_n \preceq v_1v_2\dots v_m$ si et seulement si $n \leq m$ et $\forall i \in [\![1,n]\!], u_i \leq v_i$. Si Σ comporte au moins deux lettres, cette relation d'ordre est seulement partielle. On remarquera que la restriction de $\preceq$ aux mots de Σ^d, c'est-à-dire aux tuples de longueur d fixée, correspond à l'ordre partiel étudié dans la partie précédente.

Pour les exemples, on considérera l'alphabet $\Sigma_3 = \{a, b, c\}$, avec $a < b < c$.

Pour Σ_3, on a $ab \prec bc \prec bca \prec ccab \prec ccbca \prec ccbcb$. En revanche ab, ca, bac et $aaab$, $babab$ sont deux à deux incomparables.

On note $\min(L)$ le langage des éléments minimaux de L, c'est-à-dire : $\min(L) = \{u \in L \mid \nexists v \in L, v \prec u\}$.

Par exemple, $\min(\Sigma_3^*) = \{\varepsilon\}$, $\min(\Sigma_3^+) = \{a\}$, $\min(\{ab, bc, ca, bac, ccab\}) = \{ab, ca, bac\}$.

Q22. Montrer que $\varepsilon \in L \iff \min(L) = \{\varepsilon\}$.

Il est ainsi trivial de déterminer les éléments minimaux d'un langage contenant ε. **On supposera dans toute la suite que les langages considérés ne contiennent pas ε.**

Q23. 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, sur Σ_3, les langages $L_1 = \{a^ib^j \mid (i,j) \in (\mathbb{N}^*)^2\}$ et $L_2 = \{a^ib^jc^k \mid (i,j,k) \in (\mathbb{N}^*)^3, i \leq k \wedge j \leq k\}$.

Q24. Donner $\min(L_1)$ sans justification puis montrer que les langages L_1 et $\min(L_1)$ sont réguliers.

Q25. (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 $\min(L_2)$.

Q26. Montrer que $\min(L)$ peut être régulier même si $L \subseteq \Sigma^+$ n'est pas régulier.

On se propose dans la suite de cette partie de montrer que si $L \subseteq \Sigma^+$ est régulier alors $\min(L)$ est également régulier.

III.2 – Automates finis non déterministes

Un *automate* sur un alphabet Σ est un quadruplet $\mathcal{A} = (Q, I, T, \Delta)$ avec Q un ensemble fini non vide, $I \subseteq Q$ un ensemble d'états initiaux, $T \subseteq Q$ un ensemble d'états terminaux et $\Delta \subseteq Q \times \Sigma \times Q$ une relation de transition. Un chemin d'un tel automate est une suite de transitions $((q_{i-1}, u_i, q_i))_{i \in [\![1,n]\!]} \in \Delta^n$, d'état de départ $q_0 \in Q$, d'état d'arrivée $q_n \in Q$ et d'étiquette $u = u_1 u_2 \dots u_n \in \Sigma^*$. Un mot $u \in \Sigma^*$ est reconnu par $\mathcal{A}$ s'il existe un chemin d'un état initial à un état terminal dans $\mathcal{A}$. On note $\mathcal{L}(\mathcal{A})$ le langage des mots reconnus par $\mathcal{A}$. Un exemple d'automate est représenté sur la figure 4.

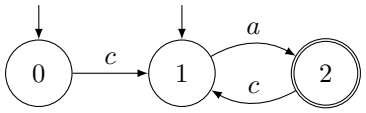

Figure 4 – L'automate $\mathcal{A}_1$ sur Σ_3. Les états initiaux, ici 0 et 1, sont indiqués par une flèche et les états terminaux, ici 2, sont doublement cerclés.

Q27. 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 $\mathcal{L}(\mathcal{A}_1)$.

III.3 – Automates marqués

Un *automate marqué* sur un alphabet Σ est un couple $\mathcal{A}^M = (\mathcal{A}, M)$ formé d'un automate $\mathcal{A} = (Q, I, T, \Delta)$ sur Σ et d'un sous-ensemble $M \subseteq \Delta$ de transitions dites *marquées*. Un mot $u \in \Sigma^*$ est reconnu par un automate marqué $\mathcal{A}^M$ s'il existe un chemin dans $\mathcal{A}$ d'un état initial à un état terminal qui passe par *au moins une* transition marquée de M. On note $\mathcal{L}(\mathcal{A}^M)$ le langage des mots reconnus par un automate marqué $\mathcal{A}^M$.

Un exemple d'automate marqué $\mathcal{A}_2^M$ est représenté sur la figure 5. Le langage $\mathcal{L}(\mathcal{A}_2^M)$ reconnu par cet automate marqué est le langage des mots de Σ_3^* d'au moins deux lettres qui commencent et terminent par a ou b et qui comportent au moins un a. On remarquera que même si l'état 0 est à la fois un état initial et un état terminal, le mot vide n'est pas reconnu par cet automate marqué, puisque l'on impose aux chemins de passer par au moins une transition marquée.

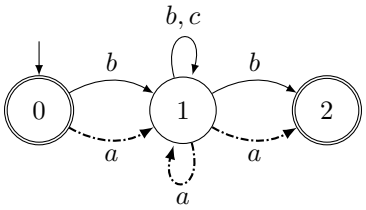

Figure 5 – Un automate marqué $\mathcal{A}_2^M$ sur Σ_3. Les états initiaux, ici seulement 0, sont indiqués par une flèche et les états terminaux, ici 0 et 2, sont doublement cerclés. Les transitions marquées, ici celles d'étiquette a, sont représentées par une ligne épaisse en traits et points.

On se propose de commencer par montrer que tout langage reconnaissable par un automate marqué est reconnaissable par un automate ordinaire.

Soit $\mathcal{A}^M = (\mathcal{A}, M)$ un automate marqué sur Σ avec $\mathcal{A} = (Q, I, T, \Delta)$. On construit un automate ordinaire $\ddot{\mathcal{A}}$ qui reconnaît le même langage que l'automate marqué $\mathcal{A}^M$. Pour cela on duplique l'automate $\mathcal{A}$, avec une première copie de $\mathcal{A}$, sans états terminaux, avec les mêmes transitions non marquées, mais dont les transitions marquées mènent vers une deuxième copie de $\mathcal{A}$, identique à $\mathcal{A}$ mais sans états initiaux.

Formellement, on note $\ddot{Q} = \{\ddot{q} \mid q \in Q\}$ une copie des états de Q. L'ensemble des états de $\ddot{\mathcal{A}}$ est $\ddot{Q} \cup Q$. L'ensemble des états initiaux de $\ddot{\mathcal{A}}$ est $\ddot{I} = \{\ddot{q} \mid q \in I\}$ et l'ensemble des états terminaux de $\ddot{\mathcal{A}}$ est T. On note $\ddot{\Delta} = \{(\ddot{q}, x, q') \mid (q, x, q') \in M\} \cup \{(\ddot{q}, x, \ddot{q'}) \mid (q, x, q') \in \Delta \setminus M\}$. Les transitions de $\ddot{\mathcal{A}}$ sont $\ddot{\Delta} \cup \Delta$.

Q28. 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.

Q29. Montrer, de manière générale, que $\mathcal{L}(\mathcal{A}^M) = \mathcal{L}(\ddot{\mathcal{A}})$.

III.4 – Automate des minimaux

Soit $L \subseteq \Sigma^+$ un langage régulier. On note $\breve{L} = \{u \in \Sigma^+ \mid \exists v \in L, v \prec u\}$ le langage des mots qui sont strictement plus grands qu'au moins un mot de L.

Q30. Montrer qu'il suffit de montrer que $\breve{L}$ est régulier pour montrer que $\min(L)$ est régulier.

On se propose de construire un automate marqué $\breve{\mathcal{A}}^M$ qui reconnaît $\breve{L}$ à partir d'un automate $\mathcal{A}$ qui reconnaît L.

Considérons un automate $\mathcal{A} = (Q, I, T, \Delta)$ qui reconnaît L.

(a) On commence par supprimer dans Δ toutes les transitions sortantes d'un état terminal de $\mathcal{A}$. Formellement, on note $\Delta' = \{(q, x, q') \mid (q, x, q') \in \Delta \text{ et } q \notin T\}$ l'ensemble des transitions de $\mathcal{A}$ qui ne partent pas d'un état terminal ;

(b) Ensuite, pour chaque transition $(q, x, q') \in \Delta'$, on ajoute (si nécessaire) et on marque les transitions (q, y, q'), pour chaque lettre $y \in \Sigma$ telle que $x < y$. Formellement, on note $M' = \{(q, y, q') \mid \exists (q, x, q') \in \Delta', \exists y \in \Sigma, x < y\}$;

(c) Enfin, on ajoute et on marque les transitions (q, x, q) pour chaque état terminal $q \in T$ et pour chaque lettre $x \in \Sigma$. Formellement on note $M'' = \{(q, x, q) \mid q \in T, x \in \Sigma\}$.

On pose $\breve{\Delta} = \Delta' \cup M' \cup M''$ et $M = M' \cup M''$. On note $\breve{\mathcal{A}} = (Q, I, T, \breve{\Delta})$ et $\breve{\mathcal{A}}^M = (\breve{\mathcal{A}}, M)$ l'automate marqué obtenu.

Q31. 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.

Q32. Montrer, de manière générale, que $\breve{L} \subseteq \mathcal{L}(\breve{\mathcal{A}}^M)$.

Q33. Montrer, de manière générale, que $\mathcal{L}(\breve{\mathcal{A}}^M) \subseteq \breve{L}$.

Q34. En déduire que si $L \subseteq \Sigma^+$ est un langage régulier alors $\min(L)$ est également un langage régulier.

IV – Meilleurs chemins dans un graphe

On s'intéresse à présent à la recherche d'itinéraires dans un réseau ferroviaire, afin d'optimiser différents critères pour un trajet. Le réseau ferroviaire est modélisé par un graphe orienté $G = (V, E)$ dont les nœuds, dans V, sont les gares. Il existe un arc d'une gare $v_1 \in V$ vers une gare $v_2 \in V$ lorsqu'il est possible de faire le trajet directement de v_1 à v_2 sans changer de train.

Chaque arc $e \in E$ est étiqueté par un couple d'entiers positifs $(t(e), p(e))$ où $t(e)$ donne le temps de trajet de la gare de départ de e vers la gare d'arrivée de e, et $p(e)$ donne le prix du billet à payer pour prendre le train représenté par e.

Un exemple d'un tel réseau est donné en figure 6. On remarquera que les trajets obtenus dans la figure 3 correspondent à certains des chemins entre le sommet $s = 0$ et le sommet $t = 5$.

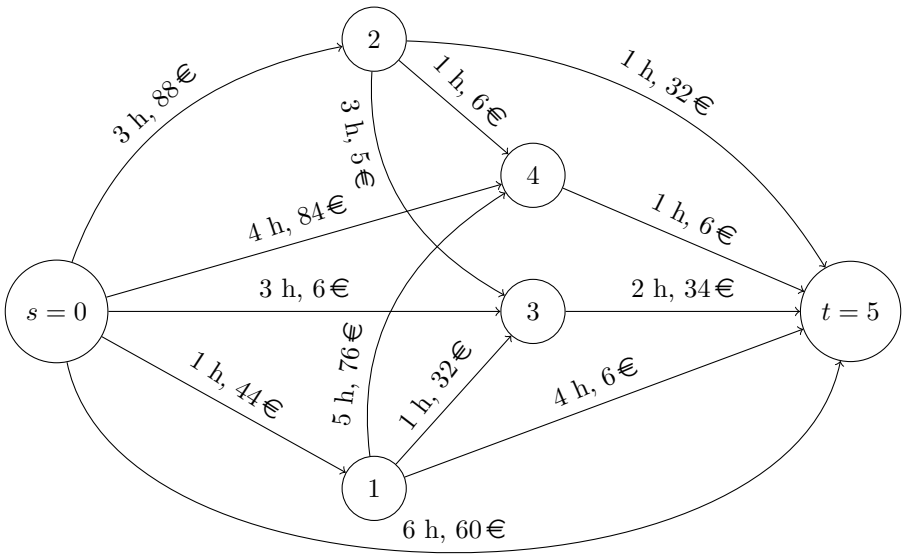

Figure 6 – Un exemple de réseau ferroviaire.

Soit $\mathcal{C} = (e_1, e_2, \cdots, e_T)$ un chemin orienté dans G, donné par la liste de ses arcs. Le *poids* de ce chemin, noté $w(\mathcal{C})$ est le triplet :

$$w(\mathcal{C}) = \left(\sum_{k=1}^{T} t(e_k), T-1, \sum_{k=1}^{T} p(e_k) \right).$$

La deuxième composante du triplet donne le nombre de correspondances, c'est-à-dire de changements de trains. Les poids sont représentés par un triplet d'entiers, qui sont ordonnés en OCaml selon l'ordre lexicographique :

```
type poids = int * int * int
```

On définit l'opération $\oplus$ sur les triplets d'entiers par : $(T, C, P) \oplus (T', C', P') = (T + T', C + C', P + P')$. On rappelle cependant que cet opérateur n'est pas défini *a priori* par le langage OCaml.

On représente un arc du graphe par un enregistrement de type `arc` :

```
type arc = {dest : int; temps : int; prix : int}
```

Ainsi, si `train` est une valeur de type `arc` représentant un arc e, l'expression `train.dest` donne le numéro de la gare d'arrivée de e, l'expression `train.temps` permet d'obtenir $t(e)$ et l'expression `train.prix` permet d'obtenir $p(e)$.

On représente les gares du réseau ferroviaire par des entiers de 0 à $n-1$, où n est le nombre total de gares. Le graphe du réseau ferroviaire est représenté par ses listes d'adjacence :

```
type reseau_ferroviaire = arc list array
```

Si `reseau` est une valeur de type `reseau_ferroviaire` et `i` un entier entre 0 et $n-1$, alors `reseau.(i)` est la liste des arcs partant du sommet `i`, dans un ordre arbitraire.

L'algorithme de Dijkstra permet, dans un graphe orienté dont les arcs sont pondérés par des entiers positifs, de calculer pour un sommet s donné les longueurs des plus courts chemins de s à tout autre sommet du graphe. On adapte cet algorithme de la façon suivante, afin qu'il calcule, dans un réseau ferroviaire, les trajets dont les poids sont des optimums de Pareto.

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 priorité (0, −1, 0)
4 : tant que Q n'est pas 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 chaque arc e de v vers w faire
10 :            Enfiler w dans Q avec la priorité π_v ⊕ (t(e), 1, p(e))
11 :        fin pour
12 :    fin si
13 : fin tant que
```

Dans cette version de l'algorithme de Dijkstra, un sommet peut être présent plusieurs fois dans la file.

Q35. 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).

On implémente le tableau `d` par un objet de type :

```
d : poids list array
```

On considère le réseau de transport de la figure 6. Après quatre itérations de la boucle de l'algorithme 1, avec $s = 0$, la file de priorité Q contient les valeurs suivantes :

Priorité	$(3,0,88)$	$(4,0,84)$	$(4,2,110)$	$(5,1,40)$	$(5,1,50)$	$(6,0,60)$	$(6,1,120)$
Sommet	2	4	5	5	5	5	4

et le tableau `d` contient les valeurs suivantes :

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

Q36. Détailler l'évolution de Q et `d` lors des 3 itérations suivantes de l'algorithme 1.

Q37. 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 pourra librement utiliser toutes les fonctions définies dans les parties précédentes, en particulier celles de la partie I.1 ainsi que celles de la partie II.4 en supposant que le type `element` correspond ici aux triplets donnant les poids.

Q38. 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.

Q39. 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 de s à t. 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.

Q40. Montrer que, dans l'algorithme 1, si π_v satisfait la condition du 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.

Q41. 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.

Q42. 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.

A – Annexe : documentation OCaml

I.1 – Fonctions usuelles

- `val min : 'a -> 'a -> 'a` renvoie le minimum de ses deux paramètres. Lorsque les paramètres sont des tuples, l'ordre utilisé est l'ordre lexicographique.
- `val max : 'a -> 'a -> 'a` renvoie le maximum de ses deux paramètres. Lorsque les paramètres sont des tuples, l'ordre utilisé est l'ordre lexicographique.

I.2 – Fonctions sur les listes

- `val List.for_all : ('a -> bool) -> 'a list -> bool`.
  L'appel `List.for_all f [a0; a1; ...; an]` renvoie `true` lorsque tous les appels `f a0`, `f a1`, ..., `f an` renvoient `true`, et `false` sinon.
- `val List.exists : ('a -> bool) -> 'a list -> bool`.
  L'appel `List.exists f [a0; a1; ...; an]` renvoie `true` lorsqu'au moins l'un des appels `f a0`, `f a1`, ..., `f an` renvoie `true`, et `false` sinon.
- `val List.filter : ('a -> bool) -> 'a list -> 'a list`.
  L'appel `List.filter f [a0; a1; ...; an]` renvoie la liste extraite de `[a0; a1; ...; an]` formée uniquement des éléments `x` tels que `f x` est `true`. Les éléments apparaissent dans la liste résultat dans le même ordre que dans la liste en paramètre.
- `val List.iter : ('a -> unit) -> 'a list -> unit`.
  L'appel `List.iter f [a0; a1; ...; an]` exécute `f a0`, puis `f a1`, etc., jusque `f an`, dans cet ordre.

I.3 – Manipulation des tableaux

- `val Array.make : int -> 'a -> 'a array`.
  L'appel `Array.make n x0` crée un tableau de `n` entrées, chacune initialisée à la valeur `x0`.
- `val Array.length : 'a array -> int` renvoie la longueur d'un tableau.
- Si `t` est un tableau et `i` un indice valide dans le tableau, `t.(i) <- x0` modifie la case d'indice `i` en lui affectant la valeur `x0`.