L'approche vorace

(Résumé du chapitre 4 du manuel)

Table des matières

4 L'approche vorace

– Note : Le terme "*glouton*" est aussi utilisé par certains auteurs plutôt que "vorace" (algorithme glouton).

– Intuition : Dans l'approche vorace pour résoudre un problème d'optimisation, une solution est obtenue en effectuant une séquence de choix, chaque choix se faisant sur la base d'information *locale*, en choisissant ce qui semble le mieux au moment où le choix s'effectue (et en "supposant" que cela va mèner, sous certaines conditions, à la solution optimale globale).

– Les algorithmes voraces sont souvent utilisés, comme les algorithmes de programmation dynamique, pour résoudre des problèmes d'optimisation. Ils produisent un résultat optimal si le problème satisfait la propriété de *choix vorace* (*greedy choice*) = la solution globale peut être obtenue par une série de choix optimums *locaux*.

Mais ce ne sont pas tous les problèmes qui satisfont cette propriété.

– Exemple : rendre la monnaie, tel qu'illustré dans l'algorithme 1.

```
DEBUT
  // Notons par montantRestant le montant qu'il reste à rendre.
  TANTQUE il reste des pièces disponibles et montantRestant ≥ 0 FAIRE
    p ← Sélectionner la pièce la plus grande parmi les pièces restantes
    SI p > montantRestant ALORS
      Rejeter la pièce p
    SINON
      Ajouter p aux pièces à rendre
      montantRestant ← montantRestant − p
    FIN
  FIN
  SI montantRestant == 0 ALORS
    TERMINER: Le problème est résolu!
  FIN
FIN
```

Algorithme 1: Algorithme pour rendre la monnaie

Exemples :

- Figure 4.1 (p. 139 du manuel) = un exemple où l'algorithme vorace 1 donne une solution optimale : on doit rendre 0.36$ avec les pièces suivantes : $1 \times 0.25\$, 2 \times 0.10\$, 1 \times 0.05\$, 2 \times 0.01\$$. L'algorithme vorace sélectionne les pièces suivantes : $0.25+0.10+0.01 = 0.36$. Trois pièces sont nécessaires, ce qui est optimal dans le cas présent.
- Figure 4.2 (p. 141 du manuel) = un exemple où l'algorithme vorace 1 *ne donne pas* une solution optimale : on doit rendre 0.16$ avec les pièces suivantes: $1 \times 0.12\$, 1 \times 0.10\$, 1 \times 0.05\$, 4 \times 0.01\$$. L'algorithme vorace sélectionne les cinq (5) pièces suivantes : $0.12 + 4 \times 0.01 = 0.16$. Or, ce n'est pas optimal car on aurait pu n'utiliser que trois (3) pièces : $0.10 + 0.05 + 0.01$.

– Principales composantes d'une approche vorace : L'algorithme débute avec un ensemble vide d'items et, de façon répétitive, ajoute un item à l'ensemble jusqu'à ce qu'une solution optimale soit obtenue. L'algorithme se compose donc, en gros, des sous-tâches suivantes, effectuées à chacune des itérations de l'algorithme :

- Une *procédure de sélection*, qui détermine le prochain item à inclure dans l'ensemble.
- Un *test de faisabilité*, qui détermine si le nouvel ensemble conduit à une solution faisable, c'est-à-dire, une solution qui peut être étendue en complétant cet ensemble par d'autres éléments.
- Une procédure de *vérification de la solution*, qui détermine si le nouvel ensemble constitue une solution à l'instance du problème.

Présentation générique des algorithmes gloutons de Brassard et Bratley

De façon plus détaillée, Brassard et Bratley (*Fundamentals of Algorithmics*, Prentice-Hall, 1996) indiquent que les algorithmes voraces possèdent généralement la plupart ou toutes les caractéristiques suivantes :

- Un problème doit être résolu de façon optimale. Pour construire la solution du problème, on a tout d'abord une collection (un ensemble, une liste, un sac, etc.) de *candidats*.
- Au fur et à mesure où l'algorithme procède, on manipule deux autres collections : une première contenant les candidats ayant été examinés et *sélectionnés*, une autre contenant les candidats ayant été examinés mais *rejetés*.
- Une fonction permet de vérifier si une collection particulière de candidats fournit une *solution* au problème, indépendamment de la question d'optimalité.
- Une autre fonction permet de vérifier si une collection de candidats est *faisable* (acceptable), c'est-à-dire, s'il serait possible ou non d'étendre cette collection de candidats de façon à obtenir une solution au problème. Ici aussi on ignore la question d'optimalité.
- Une troisième fonction, appelée *fonction de sélection*, indique à tout moment lequel des candidats restants — c'est-à-dire ceux n'ayant été ni choisis, ni rejetés — est le plus prometteur.
- Finalement, la *fonction objective* nous donne la valeur de la solution obtenue. Généralement, cette fonction n'apparaît pas de façon explicite dans l'algorithme glouton.

L'algorithme 2 présente un algorithme glouton générique basé sur la stratégie et les fonctions décrites dans les paragraphes qui précèdent.

4.1 Arbres minimum de recouvrement

Étant donné un graphe non orienté valué $G = (V, E)$ (voir Figure 4.3 du manuel, p. 139), un *arbre minimum de recouvrement* (*minimum spanning tree*, traduit aussi par *arbre couvrant minimum*) est un sous-graphe connexe de G qui contient tous les sommets de G, sous-graphe qui est aussi un arbre (graphe connexe sans cycle) *et* dont la somme des poids des arêtes est *minimale*.

Une solution vorace pour ce problème aurait l'allure du pseudo-code présenté à l'algorithme 4, où l'on cherche F tel que $T = (V, F)$ soit un arbre de recouvrement minimal pour $G = (V, E)$ (donc $F \subseteq E$).

```
PROCEDURE algorithmeGlouton( C: Collection ) S: Collection, ok: Boolean
# Entree :
#   C represente la collection des candidats.
# Sorties :
#   S va representer la collection solution, si elle existe.
#   ok va indiquer si une solution a pu etre trouvee.
DEBUT
   S ← {}
   TANTQUE C != {} ET !solution(S) FAIRE
     x ← selectionner(C)
     C ← C - {x}
     SI faisable(S U {x}) ALORS
       S ← S U {x}
     FIN
   FIN
   SI solution(S) ALORS
     ok ← VRAI
   SINON
     ok ← FAUX     # Aucune solution trouvee.
   FIN
FIN
```

Algorithme 2: Algorithme glouton générique (adapté de (Brassard et Bratley, 1996))

```
PROCEDURE solution( S: bag{Piece}, m: Montant ) r: Boolean
POSTCONDITION
  r <=> SUM( p IN S :: p ) = m

PROCEDURE selectionner( C: bag{Piece} ) pieceMax: Piece
POSTCONDITION
  pieceMax = MAXIMUM( p IN C :: p )

PROCEDURE faisable( S: bag{Piece}, m: Montant ) r: Boolean
POSTCONDITION
  r <=> SUM( p IN S :: p ) <= m

PROCEDURE rendreMonnaie( C: bag{Piece}, m: Montant ) S: bag{Piece}, ok: Boolean
POSTCONDITION
  ok => SUM( p IN S :: p ) = m & S ⊆ Cx
DEBUT
    S ← {}
    TANTQUE C != {} ET !solution(S, m) FAIRE
      x ← selectionner(C)
      C ← C - {x}
      SI faisable(S U {x}, m) ALORS
        S ← S U {x}
      FIN
    FIN
    ok ← solution(S, m)
 FIN
```

Algorithme 3: Algorithme glouton générique spécialisé pour le calcul de la monnaie

```
PROCEDURE arbreRecouvrement( G: Graphe ): Graphe
PRECONDITION
  G = (V, E)
  G est un graphe connexe
POSTCONDITION
  resultat est un arbre de recouvrement minimal pour G
DEBUT
  F ← {}
  solutionTrouvee ← FAUX
  TANTQUE NON solutionTrouvee FAIRE
    e ← Sélectionner une arête e qui semble 'intéressante'
    SI l'ajout de e à F ne crée pas de cycle ALORS
      F ← F U {e}
      SI (V, F) est un arbre de recouvrement pour G ALORS
        solutionTrouvee ← VRAI
      FIN
    FIN
  FIN
  RETOURNER (V, F)
FIN
```

Algorithme 4: Forme générale d'une solution vorace au problème de l'arbre de recouvrement minimum

Propriété cruciale d'un arbre de recouvrement minimal

Avant d'examiner plus en détails deux algorithmes différents pour la recherche d'un arbre de recouvrement minimal, il est intéressant de présenter une propriété des arbres de recouvrement qui permet de mieux comprendre et justifier pourquoi ces algorithmes fonctionnent correctement (adaptée de M.T. Goodrich et R. Tamassia, "*Data Structures and Algorithms in Java*", John Wiley & Sons, 1998). Cette propriété est illustrée à la Figure 1.

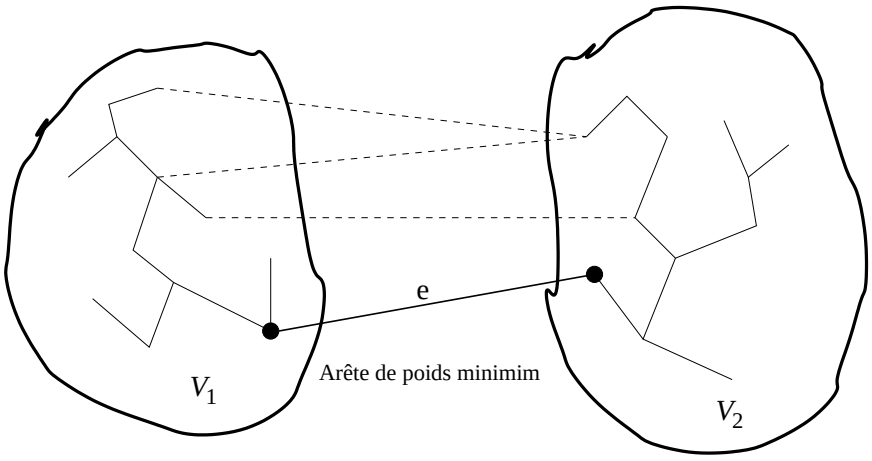

Figure 1: Une illustration de la propriété cruciale d'un arbre de recouvrement

Proposition *Soit $G = (V, E)$ un graphe non orienté, pondéré et connexe. Soit V_1 et V_2 deux ensembles disjoints non vides tels que $V = V_1 \cup V_2$. Soit e une arête dans G de poids*

minimum parmi toutes celles qui relient un sommet dans V_1 à un sommet dans V_2. Alors, il existe un arbre de recouvrement minimum T dont l'une des arêtes est e.

Justification : (Preuve par contradiction) Supposons que la proposition soit fausse, donc qu'il n'existe aucun arbre de recouvrement minimum qui contienne l'arête e (l'arête non pointillée reliant V_1 et V_2 dans la Figure 1). Soit alors T un arbre de recouvrement minimum. Si on ajoute l'arête e aux arêtes de T, un cycle sera nécessairement créé. Puisque l'ajout de e créerait un cycle, il existe donc une arête f de l'arbre T qui relie un sommet de V_1 à un sommet de V_2 (l'une des arêtes pointillées dans la Figure 1). Or, on a que le poids de e est l'arête de poids minimum parmi toutes celles reliant V_1 et V_2, c'est-à-dire, $poids(e) \leq poids(f)$. Si on supprime f de T en ajoutant e, alors on aura quand même un arbre de recouvrement, et son poids *ne sera pas supérieur* à celui de T. Puisque T était un arbre de recouvrement minimum, ce nouvel arbre obtenu par le remplacement de f par e sera lui aussi un arbre de recouvrement minimum. Or, nous avons supposé qu'il n'existait pas de tel arbre contenant e. Notre hypothèse de départ était donc fausse. On peut donc conclure qu'il existe bien un arbre de recouvrement minimum qui contienne e.

4.1.1 Algorithme de Prim

Définition 1 *Soit Y un ensemble de sommets provenant de $G = (V, E)$, G étant un graphe connexe. Le sommet* le plus près *de Y est* un *sommet $v' \in V - Y$ qui est relié à un sommet de Y par une arête de poids minimum.*

Note : S'il n'existe aucune arête entre s et s', on note alors le poids comme étant $+\infty$.

```
PROCEDURE arbreRecouvrement( G: Graphe ): Graphe
PRECONDITION
  G = (V, E)
  G est un graphe connexe
POSTCONDITION
  resultat est un arbre de recouvrement minimal pour G
DEBUT
  F ← {}
  Y ← {v1} // Le choix du sommet de départ est arbitraire.
  TANTQUE Y ≠ V FAIRE
    v' ← Sélectionner un sommet dans V−Y qui est le plus près de Y
    Y ← Y U {v'}
    e ← arête (de poids minimum) qui relie v' à Y
    F ← F U {e}
   FIN
   RETOURNER (Y, F)
FIN
```

Algorithme 5: Description de haut niveau de l'algorithme de Prim

– Description de haut niveau de l'algorithme de Prim : Algorithme 5.

– Illustration du fonctionnement de l'algorithme de Prim : Figure 2.

– Supposons que les poids des arêtes de G soient donnés par la matrice suivante :

$$W[i][j] \quad = \quad \begin{cases} poids(v_i, v_j) & \text{s'il existe une arête entre } v_i \text{ et } v_j \\ +\infty & \text{s'il n'existe aucune arête entre } v_i \text{ et } v_j \\ 0 & \text{si } i = j \end{cases}$$

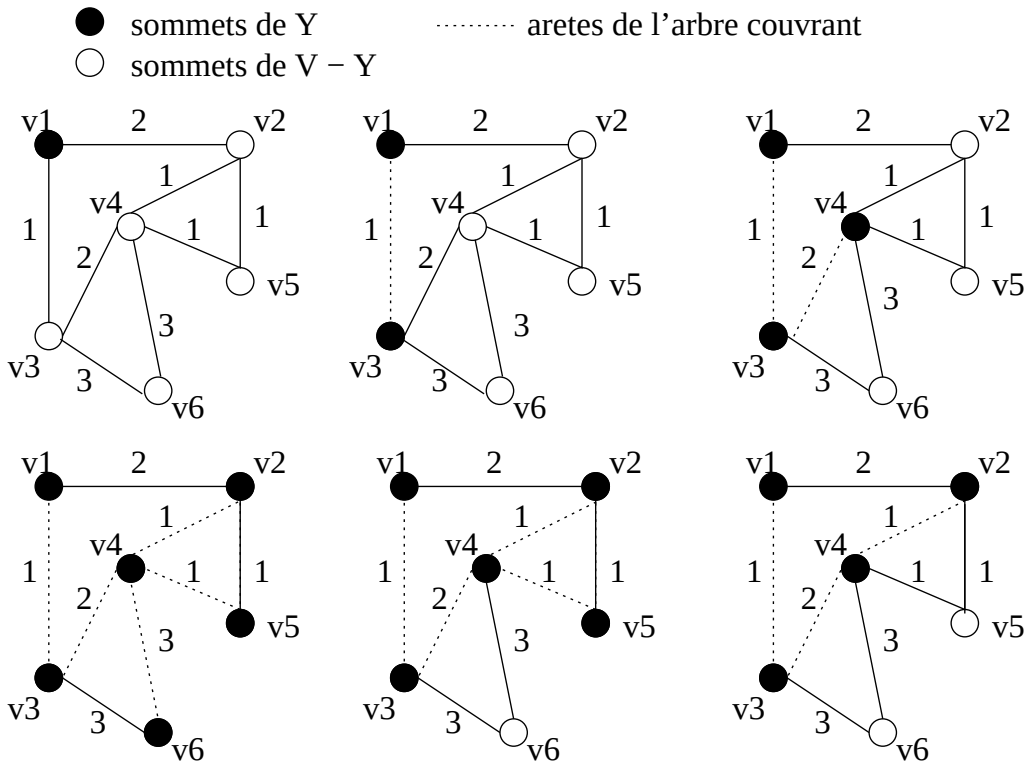

Figure 2: Illustration du fonctionnement de l'algorithme de Prim (l'arbre évolue dans le sens *horaire* à partir du coin gauche en haut)

```
PROCEDURE prim( n: Nat, W: Nat[*][*], res F: set{Arete} )
DEBUT
  F ← {}
  POUR i ← 2 A n FAIRE
    nearest[i] ← 1
    distance[i] ← W[1][i]
  FIN

  // On doit ajouter n − 1 sommets en plus du sommet initial.
  REPETER n-1 FOIS
    // On détermine le sommet qui est le plus près de Y.
    min ← +∞
    POUR i ← 2 A n FAIRE
      SI 0 ≤ distance[i] < min ALORS
        min ← distance[i]
        vnear ← i
      FIN
    FIN
    // On ajoute le sommet indexé par vnear à Y.
    e ← Arête qui relie les sommets vnear et nearest[vnear]
    F ← F U {e}
    distance[vnear] ← −1 // Distance négative indiquant que le sommet est dans Y.
    // Pour chaque sommet pas dans Y, on met à jour la distance par rapport à Y.
    POUR i ← 2 A n FAIRE
      SI W[i][vnear] < distance[i] ALORS
        distance[i] ← W[i][vnear]
        nearest[i] ← vnear
      FIN
    FIN
  FIN REPETER
FIN
```

Algorithme 6: Version pseudocode français de l'algorithme de Prim du manuel

```
PROCEDURE prim( n: Nat, W: Nat[*][*], res F: set{Arete} )
DEBUT
  DEJA_DANS_Y = −1 // Distance "bidon" indiquant un sommet déjà dans Y.

  // On initialise les structures de données.
  F ← {}
  POUR i ∈ {2,...,n} FAIRE    // On part de v1 donc 1 est ignoré.
    nearest[i] ← 1
    distance[i] ← W[1][i]
  FIN

  // On doit ajouter n − 1 sommets en plus du sommet initial v1.
  REPETER n-1 FOIS
    // On détermine le sommet qui est le plus près de Y.
    // Pour ce faire, on examine uniquement les sommets pas dans Y.
    min ← +∞
    POUR i ∈{1,...,n} TEL QUE distance[i] ≠ DEJA_DANS_Y FAIRE
      SI distance[i] < min ALORS
        min ← distance[i]
        vnear ← i
      FIN
    FIN

    // On ajoute le sommet indexé par vnear à Y.
    e ← Arête qui relie les sommets vnear et nearest[vnear]
    F ← F U {e}
    distance[vnear] ← DEJA_DANS_Y

    // Pour chaque sommet pas dans Y, on met à jour la distance par rapport à Y.
    POUR i ∈{1,...,n} TEL QUE distance[i] ≠ DEJA_DANS_Y FAIRE
      SI W[i][vnear] < distance[i] ALORS
        // Le nouveau sommet ajouté dans Y est plus près du sommet i.
        distance[i] ← W[i][vnear]
        nearest[i] ← vnear
      FIN
    FIN
  FIN REPETER
FIN
```

Algorithme 7: Version pseudocode de l'algorithme de Prim du manuel, légèrement révisée pour bien montrer que seuls les sommets *qui ne sont pas dans* Y sont examinés et traités.

L'algorithme 4.1 du manuel (pages 147–148) permet de déterminer un arbre de recouvrement minimum (le choix du sommet de départ v_1 est arbitraire). Une première version en pseudocode français est présentée à l'algorithme 6. Une deuxième version, qui rend plus explicite le fait qu'on ne traite, dans les boucles `POUR`, que les sommets qui ne sont pas déjà dans Y, est présentée à la Figure 7.

Les structures de données utilisées par l'algorithme jouent les rôles suivants :

- `nearest[i]` = indice du sommet dans Y le plus près de v_i.

  Puisque Y varie en cours d'exécution, au fur et à mesure qu'on y ajoute des sommets, ce tableau sera lui aussi mis à jour, à chaque fois en fonction du nouvel élément qui vient d'être ajouté à Y.

  Notons que, initialement, `nearest[i] = 1` pour tous les `i` ≥ 2, et ce puisque Y ne contient initialement que le sommet v_1. Soulignons que le choix du sommet initial n'a aucune d'importance : ultimement, pour que l'on ait un arbre de recouvrement, *tous les sommets* devront nécessairement avoir été inclus dans Y.

- `distance[i]` = poids de l'arête qui relie v_i au sommet indexé par `nearest[i]` (le sommet de Y le plus près du sommet v_i).

  Cette distance va évidemment être mise à jour en fonction de `nearest[i]`, c'est-à-dire, en fonction de l'ajout d'un nouvel élément dans Y.

Au fur et à mesure que de nouvelles arêtes sont ajoutées à Y, les tableaux `nearest` et `distance` sont donc mis à jour en fonction du nouveau sommet ajouté dans Y. À chaque itération de l'algorithme, il faudra donc déterminer l'index pour lequel `distance[i]` est minimum, ce qui permettra de déterminer le sommet le plus près de Y. C'est l'index de ce sommet qui est conservé dans la variable `vnear`.

Initialement, $Y = \{v_1\}$, donc ces diverses structures de données sont initialisées comme suit :

- `nearest[i] = 1`.
- `distance[i]` = poids de l'arête qui relie v_1 à v_i (peut être $+\infty$ s'il n'y a pas d'arête).

– Explications plus détaillées de l'algorithme de Prim (algorithme 6) :

- La boucle `POUR` au début de l'algorithme initialise `nearest` et `distance` pour Y ne contenant initialement que v_1.
- La première boucle `POUR` à l'intérieur de la boucle `REPETER` détermine le sommet qui est le plus près de Y. La condition $0 \leq$ `distance`[`i`] est utilisée pour ne pas examiner les sommets qui sont déjà présents dans Y (la `distance` d'un tel sommet par rapport à Y est définie, juste après l'ajout de l'arête `e` à F, comme étant `-1`).
- La deuxième boucle `POUR` met à jour les tableaux `nearest` et `distance`. On examine chacun des sommets qui n'est pas déjà dans Y (si le sommet est dans Y, alors `distance[i] =` -1, donc l'instruction `SI` n'est pas exécutée) : si la distance de ce sommet à Y est inférieure à ce qu'elle était avant l'ajout du sommet sélectionné (`vnear`), alors on modifie `distance`, en indiquant aussi que le sommet de Y qui est maintenant le plus près est bien le nouveau sommet (`nearest[i] = vnear;`).

– Complexité de l'algorithme pour $G = (V, E)$: $\Theta(n^2)$, où $n = |V|$ (nombre de sommets).

– Preuve que l'algorithme de Prim produit un arbre optimal : Voir Lemme 4.1 (p. 149 du manuel), Figure 4.6 (p. 150 du manuel) et Théorème 4.1 (p. 150 du manuel).

La preuve repose sur la notion de sous-ensemble *prometteur* (*promising*) : un sous-ensemble d'arêtes est prometteur si on peut lui ajouter des arêtes de façon à obtenir un arbre minimum de recouvrement. Tout d'abord, le lemme nous dit qu'un ensemble prometteur d'arêtes peut être étendu avec une autre arête de façon à obtenir un nouvel ensemble prometteur si on choisit une arête de poids minimum qui relie un sommet de Y à un sommet de $V - Y$, Y étant défini comme l'ensemble des sommets référés dans l'ensemble prometteur. Ensuite, il suffit de montrer par induction (Théorème 4.1) que l'ensemble F de l'algorithme est toujours prometteur.

Toutefois, sans entrer dans les détails formels de la preuve, la propriété présentée à la page 4 (Figure 1) nous donne une bonne intuition sur pourquoi l'algorithme produit bien un arbre de recouvrement minimum.

4.1.2 Algorithme de Kruskal

L'approche utilisée par l'algorithme de Kruskal est différente de celle utilisée par l'algorithme de Prim. Dans l'algorithme de Prim, on étend petit à petit l'arbre à partir d'un sommet arbitraire. À une étape donnée de l'algorithme, on a donc un ensemble de sommets reliés entre eux qui font partie de l'arbre final, et un ensemble de sommets encore isolés. Par contre, au cours de l'exécution de l'algorithme de Kruskal, on aura plutôt une *forêt* (une collection d'arbres) et, à chaque étape de l'exécution de l'algorithme, on réduira la taille de cette forêt (le nombre de sous-arbres) en reliant ensemble deux des sous-arbres déjà identifiés.

Plus précisément, dans l'algorithme de Kruskal, on commence par définir une collection de sous-ensembles disjoints de sommets (d'éléments de V). On inspecte ensuite chacune des arêtes, en ordre non décroissant de poids. Si l'arête sélectionnée relie deux sommets dans des sous-ensembles encore disjoints, alors on l'ajoute aux arêtes de l'arbre de recouvrement et on fusionne les deux sous-ensembles, puisque tous les sommets de ces deux sous-ensembles sont maintenant reliés entre eux. En d'autres mots, un sous-ensemble de sommets dénote donc un groupe de sommets qui sont actuellement reliés entre eux dans la forêt qui formera l'arbre final (une classe d'équivalence, donc). C'est pour cela que si l'arête de poids minimum relie deux sommets qui font déjà partie du même sous-ensemble, alors cette arête doit être rejetée (ne fera pas partie de l'arbre final) car son ajout créerait un cycle et on n'obtiendrait pas un arbre.

– Description de haut niveau de l'algorithme de Kruskal : Algorithme 8.

– Illustration du fonctionnement de l'algorithme de Kruskal : Figure 3.

Note : Dans l'algorithme 8, `head E'` retourne l'élément à la tête de `E'`, c'est-à-dire le premier élément de `E'` (donc `head E' = E'[1]`), alors que `tail E'` retourne la sous-séquence qui contient les mêmes éléments que `E'` sauf le premier élément (i.e., `tail E' = E'[2..length(E')]`). En fait, pour un `s` arbitraire, on a toujours que `s = [head s] ++ tail s`, ce qui peut aussi s'exprimer par la propriété suivante : `s = add(head s, tail s)`.

– Une partie clé de l'algorithme 8 est la suivante :

```
SI l'arête e relie deux sommets entre S_i et S_j avec i ≠ j ALORS
  //  On fusionne les sous-ensembles.
  S ← S − {S_i, S_j} U {S_i U S_j}
  F ← F U {e}
FIN
```

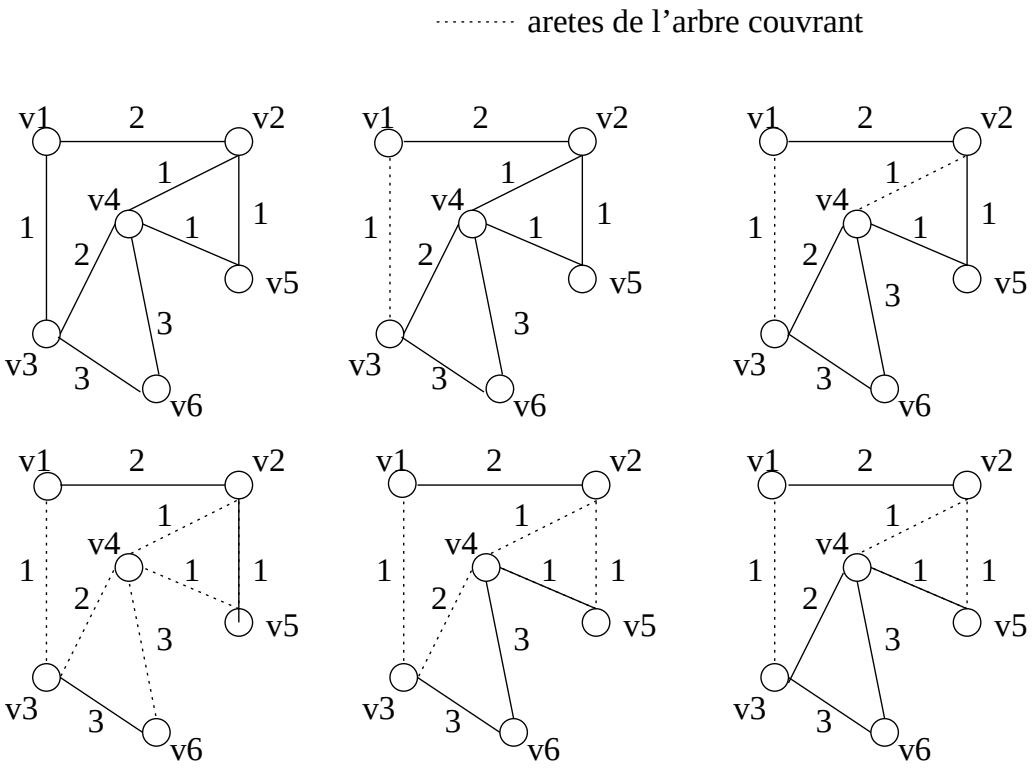

Figure 3: Illustration du fonctionnement de l'algorithme de Kruskal (la forêt évolue dans le sens *horaire* à partir du coin gauche en haut)

```
PROCEDURE arbreRecouvrement( G: Graphe ): Graphe
PRECONDITION
  G = (V, E)
  G est un graphe connexe
POSTCONDITION
  resultat est un arbre de recouvrement minimal pour G
DEBUT
  F ← {}
  // On notera S = {S_1, ..., S_k}.
  // On crée un sous-ensemble distinct pour chaque sommet.
  S ← { {v_i} | 1 ≤ i ≤ |V| }
  E' ← Séquence ordonnée (non décroissante) des arêtes provenant de E
  TANTQUE |S| ≠ 1 FAIRE
    e ← head E' // L'arête de poids minimum.
    E' ← tail E'
    SI l'arête e relie deux sommets entre S_i et S_j avec i ≠ j ALORS
      // On fusionne les sous-ensembles.
      S ← S − {S_i, S_j} U {S_i U S_j}
      F ← F U {e}
    FIN
  FIN
  S_0 ← Le sous-ensemble tel que S = {S_0}
  RETOURNER (S_0, F)
FIN
```

Algorithme 8: Description de haut niveau de l'algorithme de Kruskal

La tâche effectuée par cette partie de l'algorithme consiste à identifier les sous-ensembles $\mathtt{S}_i$ et $\mathtt{S}_j$ qui contiennent, respectivement, les sommets v_i et v_j associés à l'arête e de poids minimum sélectionnée à l'étape précèdente. Cette partie de l'algorithme pourrait donc s'exprimer de la façon suivante, en sachant que e relie les sommets d'index i et j :

```
Trouver l'ensemble S_i qui contient le sommet i
Trouver l'ensemble S_j qui contient le sommet j
SI S_i ≠ S_j ALORS
  Fusionner S_i et S_j en un seul ensemble
  F ← F U {e}
FIN
```

– L'algorithme 4.2 du manuel (présenté aux pages 151–152) permet de déterminer un arbre de recouvrement minimum utilisant l'algorithme de Kruskal. L'algorithme utilise une structure de données permettant de manipuler efficacement des sous-ensembles disjoints d'indices. Ces opérations sont décrites un peu plus bas (Annexe A). Une version en pseudocode françisé est présentée à l'algorithme 9.

– Complexité de l'algorithme pour $G = (V, E)$, où $|V| = n$ et $|E| = m$:

1. Temps pour effectuer le tri des arêtes : $\Theta(m \lg m)$.

   Note : Comme dans l'exemple du sac à dos (voir plus bas, section 4.4), on pourrait aussi utiliser une file de priorité (voir Annexe B) plutôt qu'un tri complet. La complexité asymptotique résultante serait la même, mais le facteur de proportionalité serait inférieur.

```
PROCEDURE kruskal( n: Nat, m: Nat, E: set{Arete}, res F: set{Arete} )
DEBUT
  E' ← Trier les m arêtes de E en ordre non décroissant de poids
  F ← {}
  initial( n )
  TANTQUE |F| < n-1 FAIRE
    e ← Obtenir l'arête de poids minimum de E'
    E' ← E' − {e}
    (i, j) ← indices des sommets qui sont reliés par e
    p ← find(i)
    q ← find(j)
    SI (NON equals(p, q)) ALORS
      merge(p, q)
      F ← F U {e}
    FIN
  FIN
FIN
```

Algorithme 9: Version pseudocode français de l'algorithme de Kruskal du manuel

2. Initilialisation des n ensembles disjoints à l'aide des opérations de manipulation de sous-ensembles disjoints : $\Theta(n)$.

3. Temps à l'intérieur de la boucle `TANTQUE` : on a m itérations dans le pire cas (si on doit examiner toutes les arêtes), et à chaque itération on utilise les opérations `find`, `equals` ou `merge` de manipulation de sous-ensembles disjoints. La complexité de cette partie sera donc $\Theta(m \lg^* m)$ (voir plus bas, Annexe A, p. 18 : la complexité est indépendante du nombre d'items dans l'ensemble et dépend *uniquement* du nombre d'opérations effectuées).

Parce que $m \geq n - 1$, c'est donc le tri qui domine le temps d'exécution (notons qu'on aurait le même résultat si on utilisait une file de priorité). Donc, $W(m, n) = \Theta(m \lg m)$. Toutefois, dans le pire cas, le nombre d'arêtes $m \in \Theta(n^2)$ — le pire cas est celui d'un graphe complet, c'est-à-dire d'un graphe où chaque sommet est relié à chacun des autres sommets. La complexité du pire cas peut donc aussi être exprimée par $W(m, n) \in \Theta(n^2 \lg n^2) = \Theta(n^2 \lg n)$.

– Preuve que l'algorithme de Kruskal produit un arbre optimal : Voir Lemme 4.2 (p. 154 du manuel) et Théorème 4.2 (p. 155 du manuel), preuve encore basée sur la notion d'ensemble prometteur, avec preuve par induction sur les diverses itérations. La propriété 1 peut aussi être utilisée pour justifier le fait qu'un résultat optimal est produit.

4.1.3 Comparaison entre l'algorithme de Prim et celui de Kruskal

Algorithme	**Complexité**
Prim	$T(n) \in \Theta(n^2)$
Kruskal	$W(m, n) \in \Theta(m \lg m)$ $W(m, n) \in \Theta(n^2 \lg n)$

Or, dans un graphe connexe, on a toujours la propriété suivante, où n indique le nombre de sommets et m le nombre d'arêtes :

$$n - 1 \leq m \leq \frac{n(n-1)}{2}$$

Donc :

- Si m est "petit", alors l'algorithme de Kruskal est préférable.
- Si m est "grand", alors l'algorithme de Prim est préférable.

4.2 Algorithme de Dijkstra pour les plus courts chemins

Section omise.

4.3 Ordonnancement de tâches

Section omise.

4.4 Approche vorace vs. approche de programmation dynamique : le problème du sac à dos

– Rappel : Le problème du sac à dos consiste à maximiser le "bénéfice" pouvant être obtenu en remplissant un sac avec divers items. À chaque item est associé un *bénéfice* (un entier positif) et un *poids* (un entier positif). L'objectif est de remplir le sac à dos de façon à maximiser le bénéfice, mais sans dépasser le poids maximum pouvant être contenu dans le sac.

Dans la version *fractionnaire* de ce problème, il est permis de prendre une partie d'un item (par ex., on a trois kilos de sucre et on met seulement un kilo dans le sac). Dans la version 0–1, un item doit être pris en entier ou pas du tout.

– Formulation générale du problème : on suppose qu'on a n items. Soit

$$\begin{aligned} S &= \{item_1, \ldots, item_n\} \\ w_i &= \text{poids de l'}item_i \\ p_i &= \text{bénéfice de l'}item_i \\ W &= \text{poids maximum pouvant être contenu dans le sac} \\ &\quad \text{où } w_i, p_i \text{ et } W \text{ sont des entiers positifs.} \end{aligned}$$

Le problème est de trouver un sous-ensemble $A \subseteq S$ tel que

$$\sum_{item_i \in A} p_i \text{ est maximisé sous la contrainte } \sum_{item_i \in A} w_i \leq W$$

– Solution naïve (*brute force*) : essayer tous les sous-ensembles possibles d'items et, parmi tous ceux qui satisfont la contrainte (somme des poids inférieure à W), choisir alors celui qui maximise le bénéfice. La complexité d'une telle solution serait exponentielle en fonction du nombre d'items (le nombre de sous-ensembles d'un ensemble de n éléments est 2^n).

Note : Dans ce qui suit, on supposera que la somme des poids des items est supérieure au poids total W. Autrement ... il n'y a pas de problème (on peut tout prendre).

4.4.1 Approche vorace au problème du sac à dos 0–1

Approches possibles (voir la solution pour le problème fractionnaire pour les détails) :

- Choisir en minimisant le poids
- Choisir en maximisant le bénéfice
- ...

- Choisir en maximisant le rapport bénéfice/poids (bénéfice résultant pour chaque unité de poids)

Si on utilise une approche semblable pour le problème 0–1, par exemple la dernière qui tente de maximiser la rapport bénéfice/poids, la solution résultante ne sera pas nécessairement optimale.

Exemple (Figure 4.13, p. 177 du manuel) :

- $W = 30$
- $S = \{item_1, item_2, item_3\}$
- Poids : $w_1 = 5$, $w_2 = 10$, $w_3 = 20$
- Bénéfices : $p_1 = 50$, $p_2 = 60$, $p_3 = 140$
- Rapport bénéfice/poids : $r_1 = 10$, $r_2 = 6$, $p_3 = 7$

Solution vorace vs. solution optimale :

- Solution vorace : Bénéfice résultant = 190 (poids = 25, avec les items 1 et 3).
- Autre solution possible, qui elle est optimale : Bénéfice résultant = 200 (poids 30 avec les items 2 et 3).

4.4.2 Approche vorace au problème du sac à dos fractionnaire

Puisqu'on peut prendre des fractions d'items, si on a suffisamment d'items, alors on pourra toujours remplir complètement le sac.

Solution pour l'exemple précédent : $5\ lbs \times 10\ \$/lb + 20\ lbs \times 7\ \$/lb + 5\ lbs \times 6\ \$/lb = 220\$$

Première version : Algorithme 10, qui est une version spécialisée de l'algorithme glouton générique présenté à l'algorithme 2.

Deuxième version : Algorithme 11. Cet algorithme effectue au préalable un tri des `n` items, tri qui s'effectue sur la base des quotients bénéfice/poids. Toutefois, effectuer un tel tri complet n'est pas toujours nécessaire (par exemple, si le sac ne contient, ultimement, que quelques uns des items parmi l'ensemble total).

Algorithme pour sac à dos fractionnaire avec file de priorité

L'algorithme 12 présente une version révisée de l'algorithme pour le problème du sac à dos fractionnaire, cette fois en utilisant une file de priorité (voir Annexe B).

La complexité de l'algorithme vorace résultant sera $\Theta(n \lg n)$.

4.4.3 Approche de programmation dynamique au problème du sac à dos 0–1

Voir résumé du Chapitre 3.

```
PROCEDURE poids( S: set{Nat}, poids: sequence{Nat} ) p: Nat
POSTCONDITION
  p = SUM( i IN S :: poids[i] )

PROCEDURE solution( S: set{Nat}, W: Nat, poids: sequence{Nat} ) r: Boolean
POSTCONDITION
  r <=> poids(S, poids) = W

PROCEDURE selectionner( C: set{Nat}, poids, benefs: sequence{Nat} ) i: Nat
POSTCONDITION
  benefs[i]/poids[i] = MAXIMUM( j IN C :: benefs[j]/poids[j] )


PROCEDURE Benefice Sac( W: Nat, poids, benefs: sequence{Nat} ):(set{Nat}, Nat)
PRECONDITION
 length(poids) = length(benefs) = n
 sum_{i=1}^{n} poids[i] > W
DEBUT
   C ← {1, ..., n}
   S ← {}
   benef ← 0
   TANTQUE C != {} ET !solution(S, W, poids) FAIRE
     i ← selectionner(C, poids, benefs)
     C ← C - {i}
     benef ← benef + MIN{ poids[i], W-poids(S, poids) } * benefs[i]/poids[i]
     S ← S U {i}
   FIN
   RETOURNER (S, benef)
FIN
```

Algorithme 10: Algorithme glouton générique spécialisé pour sac à dos fractionnaire

```
PROCEDURE Benefice Sac( W: Nat, poids, benefs: sequence{Nat} ):(set{Nat}, Nat)
PRECONDITION
   length(poids) = length(benefs) = n
   ∑_{i=1}^{n} poids[i] > W
DEBUT
   items ← Ordonner les items 1 à n en ordre décroissant de rapport bénéfice/poids
   benef ← 0
   S ← {}
   w ← 0
   TANTQUE w < W FAIRE
     i ← head items
     items ← tail items
     S ← S U {i}
     w' ← MIN{ poids[i], W-w }
     benef ← benef + w' * benefs[i] / poids[i]
     w ← w + w'
   FIN
   // On retourne l'ensemble des items choisis et le bénéfice total associé.
   RETOURNER (S, benef)
FIN
```

Algorithme 11: Algorithme vorace pour le problème du sac à dos fractionnaire (première version informelle)

```
PROCEDURE Benefice Sac( W: Nat, poids, benefs: sequence{Nat} ):(set{Nat}, Nat)
PRECONDITION
   length(poids) = length(benefs) = n
   ∑_{i=1}^{n} poids[i] > W
DEBUT
   ratios ← FilePriorite.creer()
   POUR i ← 1 A n FAIRE
     inserer( ratios, benefs[i]/poids[i], i )
   FIN
   benef ← 0
   S ← {}
   w ← 0
   TANTQUE w < W FAIRE
     retirerMaxElement(ratios, r, i)
     S ← S U {i}
     w' ← MIN{ poids[i], W-w }
     benef ← benef + w' * r
     w ← w + w'
   FIN
   RETOURNER (S, benef)
FIN
```

Algorithme 12: Algorithme vorace pour le problème du sac à dos fractionnaire avec file de priorité

A Manipulation de sous-ensembles disjoints (relations d'équivalence)

Certains algorithmes, par exemple, l'algorithme de Kruskal, demandent de pouvoir manipuler les sous-ensembles disjoints d'un ensemble de base. En d'autres mots, étant donné un ensemble de départ $U = \{u_1, \ldots, u_n\}$, on veut pouvoir manipuler efficacement un ensemble de sous-ensembles $U_1, \ldots, U_k$ tels que les propriétés suivantes soient satisfaites (propriétés qui définissent une *relation d'équivalence* entre les éléments, relation qui induit une *partition* des divers éléments) :

1. $U_i \neq \{\}$
2. $i \neq j \Rightarrow U_i \cap U_j = \{\}$
3. $\bigcup_{i=1}^{k} U_i = U$

De plus, on veut aussi être capable d'effectuer les opérations suivantes de façon efficace :

- Recherche : étant donné un élément u_j, on veut trouver l'ensemble U_i auquel u_j appartient (identification de la classe d'équivalence d'un élément).
- Fusion : étant donné deux sous-ensembles U_i et U_j, on veut les fusionner en un seul sous-ensemble (formation d'une nouvelle classe d'équivalence).
- Comparaison : étant donné deux sous-ensembles U_i et U_j, on veut pouvoir déterminer de façon efficace (temps constant $\Theta(1)$) si ces deux sous-ensembles sont égaux (s'ils dénotent la même classe d'équivalence).
- Initialisation : au départ, on veut faire en sorte que $U_i = \{u_i\}$ (chaque élément est dans sa propre classe d'équivalence).

Dans un tel type de données, les opérations clés ont donc l'allure suivante (version du manuel, décrite plus en détail à l'annexe C du manuel, pp. 589–598) :

- $initial(n)$: crée n sous-ensembles disjoints, chacun contenant une valeur comprise entre 1 et n (représentant donc les divers $U_i = \{u_i\}$).
- $p = find(i)$: p *réfère* à l'ensemble contenant l'élément d'index i (c'est-à-dire, la classe d'équivalence pour l'élément u_i).
- $merge(p, q)$: fusionne les deux ensembles en un seul sous-ensemble. Après l'exécution, p et q *référent* alors au même sous-ensemble.
- $equals(p, q)$: retourne `VRAI` si et seulement si p et q réfère au même sous-ensemble (dénotent la même classe d'équivalence).

La mise en oeuvre décrite dans l'annexe C du manuel utilise une structure de données appelée une *forêt d'arbres inversées* (un enfant pointe vers son parent, alors la racine pointe simplement vers elle-même). Une telle structure peut être réalisée à l'aide d'un tableau d'items, donc pas nécessairement avec allocation dynamique et pointeurs (comme pour un monceau, sauf que l'organisation des items est différente).

Les détails de la mise en oeuvre d'une telle structure de données dépassent le cadre du présent cours (voir l'exemple présenté en classe ; une présentation plus détaillée serait plutôt le sujet du cours INF7341 Structures de données). On peut toutefois mentionner qu'une mise en oeuvre naïve peut conduire à des résultats tels qu'une opération de recherche soit de temps linéaire. Par contre, en s'assurant par diverses techniques (fusion qui tient compte de la taille des ensembles à fusionner et compression des chemins lors d'une recherche) de minimiser le plus possible la hauteur des arbres inversés manipulés, on peut en arriver à une mise en oeuvre plus efficace. Plus précisément, pour une série de m opérations *equals*, *find* ou *merge*, la complexité peut être $O(m \lg^* m)$, où la fonction $\lg^* m$ est définie comme suit :

$$\begin{aligned} \lg^* m &= min\{i : t(i) \geq m\} \\ t(i) &= \begin{cases} 1 & \text{si } i = 0 \\ 2^{t(i-1)} & \text{si } i \geq 1 \end{cases} \end{aligned}$$

L'analyse détaillée de la complexité de cette structure de données et des opérations associées requiert l'utilisation d'une technique appelée *analyse amortie*. Dans une telle forme d'analyse, on s'intéresse non pas strictement à la complexité d'*une* opération *donnée*, mais plutôt à la complexité *d'une série d'opérations*. Intuitivement, ceci signifie qu'on peut permettre qu'une opération donnée soit un peu plus coûteuse si on est assuré que ce coût additionnel pourra être *amorti* sur l'ensemble des opérations. Dans le cas présent, la borne asymptotique ne porte donc pas sur un appel spécifique à une opération donnée, mais porte plutôt sur le coût pour exécuter (après l'appel à *initial*) une *série* de m opération *equals*, *find* ou *merge*, la borne étant alors de $O(m \lg^* m)$.

Note : Dans le livre de Cormen, Leiserson et Rivest, la borne mentionnée est plutôt $O(m \lg^* n)$. Toutefois, la construction de l'ensemble initial se fait plutôt à l'aide de n appels à une fonction d'initialisation pour créer un ensemble singleton (plutôt qu'avec un unique appel à *initial*) et le nombre m d'opérations inclut aussi ces n appels d'initialisation de l'ensemble. Le résultat est donc équivalent. Notons aussi que l'analyse de la version plus efficace demande plusieurs pages (8) d'analyse mathématique détaillée, et ce *après* que les notions de base de l'analyse amortie aient déjà été présentées dans un chapitre précédent ...

Note : Dans le manuel (appendice C), la borne mentionnée est plutôt $O(m \lg m)$. Le dernier paragraphe de l'annexe C mentionne toutefois l'existence de la technique de *path compression*, qui permet d'obtenir une mise en oeuvre qui soit presque de temps linéaire en m (le nombre d'opérations). Cette mise en oeuvre avec *path compression* est bien celle mentionnée dans les paragraphes qui précèdent. Pour obtenir la borne moins stricte $O(m \lg m)$, il suffit simplement de fusionner les arbres inversés de façon à minimiser la hauteur de l'arbre résultant, ce qui se fait simplement en faisant pointer la racine du plus petit arbre vers la racine du plus grand.

B Files de priorité et opérations associées

De nombreux problèmes demandent de manipuler des ensembles d'éléments et de pouvoir associer à ces différents éléments un *niveau de priorité*. Lorsqu'un item doit être retiré de l'ensemble, c'est alors le niveau de priorité associé aux divers éléments qui détermine lequel sera obtenu. Une file de priorité ne doit donc pas être confondue avec une *file FIFO*, où l'ordre de retrait des éléments se fait simplement dans l'ordre d'arrivée (*First-In, First-Out*).

– Les opérations de manipulation d'une file de priorité sont les suivantes :

- `estVide(fp)` : la file de priorité `fp` est-elle vide?
- `inserer(fp, c, e)` : insère l'élément `e` ayant la clé `c` (niveau de priorité) dans la file `fp`.

- `maxElement(fp)` : retourne l'élément dont la clé possède la valeur (pour lequel la priorité est) maximale.
- `maxCle(fp)` : retourne la clé dont la valeur est maximale.
- `retirerMaxElement(fp, cle, elem)` : supprime de la file `fp` l'élément dont la clé possède la valeur maximale et retourne cet élément et la clé qui lui est associée.

```
type FilePriorite = ptr ...;

procedure creer() returns FilePriorite fp
# POSTCONDITION
#   new(fp)
#   fp = {}

procedure estVide( FilePriorite fp ) returns bool r
# POSTCONDITION
#   r <=> fp = {}

procedure inserer( FilePriorite fp, int cle, int elem )
# POSTCONDITION
#  fp = fp' U {(cle, elem)}

procedure maxCle( FilePriorite fp ) returns int cle
# PRECONDITION
#   ~estVide(fp)
# POSTCONDITION
#   cle = MAXIMUM( (c, e) IN fp :: c )

procedure maxElement( FilePriorite fp ) returns int elem
# PRECONDITION
#   ~estVide(fp)
# POSTCONDITION
#   SOME( cle ::  (cle, elem) IN fp & cle = maxCle(fp) )

procedure retirerMaxElement( FilePriorite fp, res int cle, res int elem )
# PRECONDITION
#   ~estVide(fp)
# POSTCONDITION
#   (cle, elem) IN fp' & cle = maxCle(fp')
#   fp = fp' - {(cle, elem)}
```

Type abstrait de données 1: Types et procédures pour la manipulation de files de priorité (mise oeuvre omise)

Les interfaces de ces opérations peuvent être décrites de façon un peu plus formelle (avec une syntaxe de style MPD pour les déclarations, donc transmission de la file à manipuler comme premier argument, et avec une syntaxe de style `Spec` pour les pré/post-conditions) telles qu'illustrées dans la spécification du type abstrait 1.

– Un exemple d'utilisation d'une file priorité est présenté à l'algorithme 13, qui effectue un tri d'une séquence d'éléments en utilisant une file de priorité. Il est important de noter que la complexité de l'algorithme de tri résultant va alors dépendre *de la mise en oeuvre choisie pour la file de priorité.*

```
PROCEDURE trierAvecFilePriorite( items: sequence{Item} ): sequence{Item}
DEBUT
   // On suppose qu'on doit trier en ordre croissant des clés (cle) associées aux items.
   fp ← FilePriorite.creer()
   POUR k ← 1 A length(items) FAIRE
     inserer( fp, cle(items[k]), items[k] )
   FIN
   resultat ← []
   TANTQUE NON estVide(fp) FAIRE
     retirerMaxElement(fp, c, item)
     resultat ← add(item, resultat)
   FIN
   RETOURNER resultat
FIN
```

Algorithme 13: Tri d'une séquence d'items à l'aide d'une file de priorité

1. Complexité (pire cas) des opérations de manipulation d'une file de priorité réalisée avec une *séquence non ordonnée* d'éléments (par ex., tableau alloué de façon statique) :
   - Déterminer si la file est vide : $\Theta(1)$.
   - Identifier l'élément ou la clé maximal : $\Theta(n)$.
   - Insérer un item dans la file : $\Theta(1)$.
   - Retirer l'élément ayant la clé maximale de la file : $\Theta(n)$.

   Soit n la longueur de la séquence à trier (`items`). L'algorithme de tri 13 serait alors, dans le pire cas, de complexité $\Theta(n^2)$.

2. Complexité (pire cas) des opérations de manipulation d'une file de priorité réalisée avec une *séquence ordonnée* d'éléments (par ex., liste chaînée avec allocation dynamique) :
   - Déterminer si la file est vide : $\Theta(1)$.
   - Identifier l'élément ou la clé maximal : $\Theta(1)$.
   - Insérer un item dans la file : $\Theta(n)$.
   - Retirer l'élément ayant la clé maximale de la file : $\Theta(1)$.

   Soit n la longueur de la séquence à trier (`items`). L'algorithme de tri 13 serait alors, dans le pire cas, de complexité $\Theta(n^2)$.

3. Complexité (pire cas) des opérations de manipulation d'un file de priorité mise en oeuvre avec un *monceau* (*heap*) :
   - Déterminer si la file est vide : $\Theta(1)$.
   - Identifier l'élément ou la clé maximal : $\Theta(1)$.
   - Insérer un item dans la file : $\Theta(\lg\, n)$.
   - Retirer l'élément ayant la clé maximale de la file : $\Theta(\lg\, n)$.

   Soit n la longueur de la séquence à trier (`items`). L'algorithme de tri 13 serait alors, dans le pire cas, de complexité $\Theta(n\, \lg\, n)$.