Une boîte d'outils pour la programmation en Caml

Laurent Chéno
avec la collaboration de
Alain Bèges

été 1995

Lycée Louis-le-Grand, Paris

Table des matières

Liste des figures

Liste des programmes

Introduction

Ce poly comprend trois parties : la première, intitulée *Structures de données*, décrit des structures qu'on rencontre dans toutes sortes d'algorithmes classiques. On peut voir les programmes Caml écrits pour les implémenter comme des briques de base de plus gros programmes, comme une librairie de fonctions, pour parler davantage comme un informaticien.

dans la suite, nous préfèrerons le terme bibliothèque *à* librairie*, qui n'est qu'une pauvre traduction de l'anglais* library

Ensuite on trouve une partie sur les *graphes*, où nous exposons — et implémentons en Caml — quelques uns des algorithmes les plus classiques sur les graphes.

Une dernière partie s'intéresse à ce qu'on appelle communément la géométrie combinatoire. Il s'agit de détailler quelques algorithmes classiques de géométrie comme par exemple la recherche de l'enveloppe convexe d'un ensemble de points du plan.

Nous avons fait figurer en annexe un petit manuel de référence — écrit en collaboration avec A. Bèges — qui ne prétend pas se substituer aux deux ouvrages de X. Leroy et P. Weis.

maîtriser ou apprivoiser. . .

On trouvera également en annexe quelques indications de programmation en Caml pour maîtriser le style fonctionnel, ce qui sera d'autant plus utile qu'on aura déjà travaillé dans des langages comme `C` ou Pascal.

Enfin on fournit un petit glossaire dans les deux sens anglais vers français et inversement.

Pour terminer cette petite introduction, je voudrais remercier Bruno Petazzoni, qui a bien voulu assurer la corvée de la relecture de tout ce texte, et qui a relevé une collection de fautes d'orthographe ou de typographie à faire rougir mes instituteurs.

Partie I

Structures de données

Chapitre 1

Piles

1.1 Définition abstraite de la structure

pile ≡ stack

Une pile est une structure qui autorise les trois opérations fondamentales suivantes :

- l'empilage d'un objet, c'est-à-dire son insertion dans la structure ;
- le dépilage du dernier objet qui a été empilé dans la structure ;
- le test qui détermine si la pile est vide ou non.

En anglais on dira *push* pour empiler, et *pop* pour dépiler.

On ajoute parfois une procédure qui permet simplement de visiter le sommet de pile, c'est-à-dire l'objet qui serait dépilé au moment considéré. Il faut noter que les procédures précédentes permettent d'obtenir le même résultat : pour visiter le sommet de pile, on dépile le sommet de pile, on le recopie sur son cahier, et on le rempile pour laisser la pile dans l'état où on l'avait trouvée en entrant.

Les Américains utilisent souvent l'expression *LIFO*, acronyme de *Last In, First Out*, pour illustrer que c'est toujours l'objet qui a été empilé en dernier qui est dépilé en premier, ou, si l'on préfère, que l'on procède comme avec une pile d'assiettes : c'est toujours la dernière assiette posée (tout en haut de la pile) qu'on retire la première.

sauf peut-être chez M. Tex Avery

1.2 Une structure concrète

Il n'est pas difficile d'imaginer une structure simple pour implémenter les piles. Il suffit en effet d'une liste : empiler un objet `obj` dans la pile `pile` revient tout bêtement à remplacer la pile par `obj :: pile` ; dépiler est un simple appel à la fonction `hd` (*head*) de la bibliothèque standard, qui renvoie le premier élément de la liste.

Nous allons ici décrire une structure plus complexe. Pourquoi?

Il est vrai que Caml se charge tout seul de gérer sa mémoire dans le cas de la gestion des listes. On ne peut cependant ignorer que les listes Caml sont des listes chaînées, et donc que chaque cellule de liste utilise la mémoire nécessaire à l'objet même qu'elle contient mais aussi à un lien (un pointeur, si on veut) vers la cellule suivante de la liste. On peut essayer — quitte à compliquer la structure et donc les opérations qui la gèrent — d'avoir un meilleur contrôle sur cette gestion de la mémoire.

un pointeur, ça crée des liens

On propose donc d'utiliser une liste chaînée non pas de cellules contenant 1 objet, mais de tableaux d'objets de taille fixe (des *blocs*). Au moment d'empiler, ou bien on a encore assez de place dans les différents blocs alloués, et c'est tout bon, ou bien on manque de place, et on alloue un nouveau bloc de mémoire sous la forme d'un nouveau tableau.

Bien entendu, il faut gérer la liste de blocs et noter la position dans ce bloc du sommet de pile. C'est le prix à payer pour maintenir la structure. D'un autre côté si chaque bloc comporte

k objets, on n'aura plus besoin que d'environ n/k pointeurs au lieu des n de la structure naïve.

1.3 Une implémentation en Caml

La pile utilisera une liste des blocs alloués, que nous appellerons `liste_des_blocs_alloués` ; deux entiers, l'indice du sommet de pile dans son bloc, noté `indice_dans_bloc`, et le nombre de blocs qu'on a déjà alloués depuis la naissance de la pile, `nb_blocs_réservés`.

Au début la liste des blocs est vide, le nombre de blocs alloués est nul, et, conventionnellement, l'indice dans le bloc courant $k-1$, si k est la taille de chaque bloc alloué.

Pour dépiler, on déclenchera une exception `Pile_Vide` si le nombre de blocs alloués est nul. Sinon, les variables définies ci-dessus permettent de récupérer directement le sommet de pile. Reste à les mettre à jour, en décrémentant l'indice dans le bloc, sauf s'il était nul : alors il faudrait aussi retrancher le bloc courant de la liste des blocs, et positionner l'indice dans le bloc du sommet de pile à $k-1$. La consultation du sommet de pile se fait de façon analogue.

Pour empiler, si l'indice dans le bloc est au plus égal à $k-2$, on se contente de l'incrémenter et de ranger l'objet à empiler dans la case désignée du bloc. Sinon on crée un nouveau bloc qu'on ajoute à la liste.

Chaque nouvelle pile doit gérer sa propre liste de blocs, et ses propres index : le nombre de blocs alloués, la position courante dans le bloc. On ne veut évidemment pas inonder le monde Caml de variables globales en trop grand nombre, ni d'ailleurs permettre à l'utilisateur de modifier les variables de la pile, sous peine de se retrouver dans des situations périlleuses. On va donc *encapsuler* les différents éléments nécessaires à la gestion de la pile dans la pile elle-même. Comme on aura ainsi interdit à l'utilisateur l'accès aux variables de la pile, la fonction `crée_pile` qu'on va écrire renverra non pas un pseudo-objet de type pile, mais *les fonctions de gestion de la pile*, sous la forme d'un quadruplet `(dépiler,empiler,visiter,test_à_vide)` de fonctions. `crée_pile` aura deux arguments : le premier sera un objet du type de ceux que l'on veut empiler (Caml l'exige pour son typage), le second la taille des blocs alloués, en nombre d'objets.

ce texte est imprimé dans la police Apollo. . .

On trouvera ci-dessous l'interface correspondante, et, plus loin, un exemple d'utilisation de la structure dans une session type.

Programme 1.1 L'interface `Piles.mli`

```
1 exception Pile_Vide ;;
2
3 value crée_pile : 'a -> int -> (unit -> 'a) * ('a -> unit) * (unit -> 'a) * (unit ->
  bool) ;;
4
5 (* (dépiler,empiler,visiter,test_à_vide) *)
```

Montrons maintenant comment on peut, en Caml, *encapsuler* constantes et variables dans une procédure. Ce procédé, très général, sera régulièrement utilisé tout le long de nos exemples. Il permet en quelque sorte de mimer (et en mieux, dans un certain sens) la programmation objet qu'on retrouve dans d'autres langages de programmation.
Nous écrirons quelque chose du genre :

```
let procédure_englobante arg1 arg2 =
    let var1 = ref init1 and var2 = ref init2 and ...
    and k1 = expr1 and k2 = expr2 and ...
    in
    let f1 x y = ...
    and f2 w = ...
    in (f1,f2,...) ;;
```

Nous assurons ainsi que seules les procédures `f1`, `f2` ont accès aux variables `var1`, `var2`, et aux constantes `k1`, `k2`. Remarquons la différence entre les références dont la valeur pourra

n'est pas mutable *qui veut*

être modifiée et les constantes qui ne correspondent qu'à des définitions. Notons enfin que les arguments `arg1` et `arg2` de la procédure englobante jouent un peu le même rôle que `k1` et `k2` pour les fonctions `f1` et `f2`.

Programme 1.2 Une session exemple : utilisation des piles

```
1 >       Caml Light version 0.6
2
3 #load_object "Piles.zo";;
4 - : unit = ()
5 ##open "Piles";;
6 #let (pop,push,scan,vide) = crée_pile "bidon" 10;;
7 vide : unit -> bool = <fun>
8 scan : unit -> string = <fun>
9 push : string -> unit = <fun>
10 pop : unit -> string = <fun>
11 #vide();; (* la pile est-elle vide ? *)
12 - : bool = true
13 #do_list push ["a";"b";"c";"d";"e";"f";"g";"h";"i";"j";"k";"l";"z"];;
14 - : unit = ()
15 #scan();; (* visite du sommet de pile *)
16 - : string = "z"
17 #scan();; (* toujours le même *)
18 - : string = "z"
19 #pop();; (* là on le dépile ! *)
20 - : string = "z"
21 #scan();; (* la preuve : *)
22 - : string = "l"
23 #push "m";; push "n";;
24 - : unit = ()
25 #- : unit = ()
26 #pop();; pop();; pop();; pop();; pop();;
27 pop();; pop();; pop();; pop();; pop();;
28 - : string = "n"
29 #- : string = "m"
30 #- : string = "l"
31 #- : string = "k"
32 #- : string = "j"
33 #- : string = "i"
34 #- : string = "h"
35 #- : string = "g"
36 #- : string = "f"
37 #- : string = "e"
38 #pop();; pop();; pop();; pop();;
39 - : string = "d"
40 #- : string = "c"
41 #- : string = "b"
42 #- : string = "a"
43 #pop();;
44 Uncaught exception: Pile_Vide
45 #
```

Dans le cas qui nous intéresse, nous encapsulons les trois variables explicitées ci-dessus, `liste_des_blocs_alloués`, `nb_blocs_réservés` et `indice_dans_bloc`. Le dépilage commence par la sélection du sommet de pile `a` qu'on trouvera dans le premier bloc de la liste d'allocation, et continue par la mise à jour des variables de la pile. La consultation du sommet de pile est encore plus simple. Enfin l'empilage pourra en cas de besoin invoquer la fonction `make_vect` pour allouer un nouveau bloc de mémoire.

On trouvera l'intégralité du fichier `Piles.ml` ci-dessous.

Remarque : une fois compilée l'interface `Piles.mli`, on dispose d'un fichier `Piles.zi`, et il n'est plus utile de redéclarer l'exception `Pile_Vide`. C'est ce qui explique le commentaire en ligne 1.

Programme 1.3 `Piles.ml`

```
(*  exception Pile_Vide;; *)

(* crée_pile attend deux arguments: le premier n'a d'utilité        *)
(* que pour fixer le type des éléments de la pile                   *)
(* le second est un entier décrivant la taille des blocs            *)
(* d'allocation mémoire successivement créés                        *)
(* si on s'attend à une taille maximale (à peu près) de N           *)
(* il paraît intelligent de la signaler lors de cet appel           *)
(* la fonction renvoie les fonctions standard d'accès à             *)
(* une pile : à savoir  le dépilage     : pop       : unit -> 'a    *)
(*                      la consultation : scan      : unit -> 'a    *)
(*                      l'empilage      : push      : 'a -> unit    *)
(*                      le test à vide  : est_vide  : unit -> bool  *)

let crée_pile x k =
    if k <= 0 then failwith "crée_pile objet_type taille_du_bloc attend une
taille_du_bloc > 0"
    else
    let liste_des_blocs_alloués = ref ([ ])
    and nb_blocs_réservés = ref 0
    and indice_dans_bloc = ref (k-1)
    in
    let pop () =
        if !nb_blocs_réservés = 0 then raise Pile_Vide
        else
        begin
            let a = (hd !liste_des_blocs_alloués).(!indice_dans_bloc)
            in
            if !indice_dans_bloc = 0 then
            begin
                indice_dans_bloc := k-1;
                nb_blocs_réservés := !nb_blocs_réservés - 1;
                liste_des_blocs_alloués := tl !liste_des_blocs_alloués
            end
            else indice_dans_bloc := !indice_dans_bloc - 1;
            a
        end
    and scan () =
        if !nb_blocs_réservés = 0 then raise Pile_Vide
        else (hd !liste_des_blocs_alloués).(!indice_dans_bloc)
    and push a =
        if !indice_dans_bloc < k-1 then
        begin
            indice_dans_bloc := !indice_dans_bloc + 1;
            (hd !liste_des_blocs_alloués).(!indice_dans_bloc) <- a
        end
        else
        begin
            let nouveau_bloc = make_vect k x
            in
            nb_blocs_réservés := 1 + !nb_blocs_réservés;
            liste_des_blocs_alloués := nouveau_bloc :: !liste_des_blocs_alloués;
            indice_dans_bloc := 0;
            nouveau_bloc.(0) <- a
        end
    and est_vide () = !nb_blocs_réservés = 0
    in (pop,push,scan,est_vide);;
```

Chapitre 2

Files d'attente

2.1 Définition abstraite de la structure

Une file d'attente, on dit parfois aussi une queue (y compris en anglais, d'ailleurs), est une structure de données qui permet les opérations suivantes :

- l'insertion d'un nouvel objet dans la structure ;
- la suppression de l'objet inséré le premier dans la structure ;
- le test qui détermine si la file est vide ou non.

On ajoutera éventuellement une fonction qui renvoie la liste des éléments de la file.

On aura compris pourquoi cette structure s'appelle une file d'attente : comme dans la queue devant un guichet, le premier arrivé est celui qui s'en va le premier. On traite les personnes dans l'ordre où elles se sont présentées. Il s'agit donc d'une structure de type *FIFO : First In, First Out*, quand la structure de pile était du type *LIFO*.

2.2 Interface proposée

Caml veut des types !

D'une façon analogue à ce qui a été fait pour les piles, nous écrirons une fonction que nous appellerons `crée_file_d_attente` et qui prend en argument un objet du type de ceux qu'on veut utiliser dans la file et qui renvoie un quadruplet de *fonctions d'accès* à la structure : `(ajoute, extrait, est_vide,en_liste)` qui respectivement insère un objet, extrait l'objet le plus ancien, teste si la file est vide, et rend la liste des objets présents.

Voici l'interface correspondante :

Programme 2.1 L'interface `Files_d_attente.mli`

```
1 exception File_Vide ;;
2 value crée_file_d_attente : 'a -> ('a -> unit) * (unit -> 'a) * (unit -> bool) *
  (unit -> 'a list) ;;
```

2.3 Une structure concrète

Une première idée pour implémenter les files d'attente consiste à utiliser une liste : on insèrera par exemple les nouveaux objets en tête de liste, et on trouvera le plus ancien en queue de liste. Évidemment, ce n'est pas si simple, car pour trouver la queue de liste il faudra parcourir toute la liste, et supprimer cet élément semble peu évident. Inversons donc la question : on

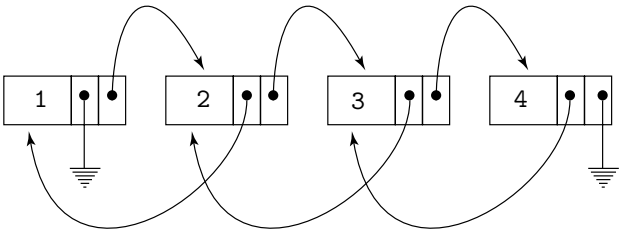


Figure 2.1: La structure de liste doublement chaînée

insère en queue de liste et on supprime en tête. Cette fois c'est la suppression qui est évidente. En revanche insérer un objet en queue de liste demande à nouveau un parcours complet de la liste.

Bref, la structure de liste que nous offre Caml semble mal adaptée à notre problème.

Nous allons construire une structure plus appropriée : les listes doublement chaînées.

Il s'agit de listes formées de cellules comprenant un objet de la liste, et des liens sur les cellules précédente et suivante, quand elles existent, ou sur `Nil` sinon. Un petit schéma (voir ci-dessus) explique la structure d'une liste doublement chaînée contenant les entiers 1, 2, 3 et 4.

`Nil` *est à la terre*

Voici la déclaration du type correspondant en Caml :

```
type 'a liste_doublement_chaînée = Nil | Cellule of 'a cellule_de_liste_doublement_chaînée
and 'a cellule_de_liste_doublement_chaînée =
        {    mutable valeur       : 'a ;
             mutable lien_arrière : 'a liste_doublement_chaînée ;
             mutable lien_avant   : 'a liste_doublement_chaînée       } ;;
```

La file d'attente sera représentée par une liste doublement chaînée, avec un accès rapide grâce à deux pointeurs supplémentaires, l'un sur la première cellule, l'autre sur la dernière de la liste. Ainsi notre type s'écrira-t-il :

```
type 'a file_d_attente = {  mutable tête  : 'a liste_doublement_chaînée ;
                            mutable queue : 'a liste_doublement_chaînée   } ;;
```

On procèdera à l'insertion d'un nouvel objet en tête de la liste doublement chaînée, la suppression se réalisant donc sur la fin de la liste, à laquelle on accède directement grâce au pointeur `queue`. La seule difficulté est dans la gestion des différents pointeurs, mais elle se surmonte sans peine grâce à quelques schémas.

l'habitude est de parler de l'insertion, ou de la suppression d'une clé *dans une structure*

On trouvera le programme final page suivante.

Notons que l'insertion comme la suppression se réalisent en temps constant : le coût est $O(1)$.

Programme 2.2 `Files_d_attente.ml`

```
type 'a liste_doublement_chaînée = Nil | Cellule of 'a cellule_de_liste_doublement_chaînée
and 'a cellule_de_liste_doublement_chaînée =
            {   mutable valeur : 'a ;
                mutable lien_arrière : 'a liste_doublement_chaînée ;
                mutable lien_avant : 'a liste_doublement_chaînée    } ;;

type 'a file_d_attente = {  mutable tête : 'a liste_doublement_chaînée ;
                            mutable queue : 'a liste_doublement_chaînée } ;;

let crée_file_d_attente (bidon:'a) =
    let (file:'a file_d_attente) = { tête = Nil ; queue = Nil }
    in
    let ajoute a =
        let c = Cellule { valeur = a ; lien_arrière = Nil ; lien_avant = file.tête }
        in
        match file.tête with
              Nil -> file.tête <- c ; file.queue <- c
            | Cellule ct -> ct.lien_arrière <- c ; file.tête <- c
    and extrait () =
        match file.queue with
              Nil -> raise File_Vide
            | Cellule cq -> match cq.lien_arrière with
                      Nil ->    file.tête <- Nil ;
                                file.queue <- Nil ;
                                cq.valeur
                    | (Cellule cp) as p ->  file.queue <- p ;
                                            cp.lien_avant <- Nil ;
                                            cq.valeur
    and est_vide () = file.tête = Nil
    and en_liste () =
        let rec listeur = function
              Nil -> []
            | Cellule c -> c.valeur:: (listeur c.lien_avant)
        in
        listeur file.tête
    in
    (ajoute,extrait,est_vide,en_liste) ;;
```

Chapitre 3

Une structure pour les listes circulaires

3.1 Description de la structure

Dans ce court chapitre, nous écrivons en Caml de quoi implémenter des listes circulaires doublement chaînées. Il s'agit d'une structure ordonnée, dans laquelle nous pouvons avancer comme reculer, et qui boucle sur elle-même : si la structure comporte les n objets x_0, x_1, ..., x_{n-1}, il est défini sur toute la structure une fonction *prédécesseur* et une fonction *successeur*, telles que

tu as remarqué ? on travaille modulo n

$$\begin{aligned} \forall i > 0, \quad & \textit{prédécesseur}(x_i) = x_{i-1}; \\ & \textit{prédécesseur}(x_0) = x_{n-1}; \\ \forall i < n-1 \quad & \textit{successeur}(x_i) = x_{i+1}; \\ & \textit{successeur}(x_{n-1}) = x_0. \end{aligned}$$

Comme on le verra aisément, nous nous inspirerons largement de ce qui a été vu dans le chapitre précédent : nous utiliserons ici encore une liste doublement chaînée.

3.2 Implémentation en Caml

remâcher trop ne donne qu'une infâme bouillie

Ayant largement explicité précédemment l'utilisation des listes doublement chaînées, nous nous contenterons ici de fournir les listages des fichiers sources.

Programme 3.1 L'interface `Cycles.mli` pour les listes circulaires

```
1 type 'a cycle_bien_chaîné;;
2
3 exception Cycle_Vide;;
4
5 value       insère_devant : 'a -> 'a cycle_bien_chaîné -> 'a cycle_bien_chaîné
6 and       insère_derrière : 'a -> 'a cycle_bien_chaîné -> 'a cycle_bien_chaîné
7 and          valeur_cycle : 'a cycle_bien_chaîné -> 'a
8 and          avance_cycle : 'a cycle_bien_chaîné -> 'a cycle_bien_chaîné
9 and          recule_cycle : 'a cycle_bien_chaîné -> 'a cycle_bien_chaîné
10 and supprime_valeur_cycle : 'a cycle_bien_chaîné -> 'a cycle_bien_chaîné
11 and        liste_en_cycle : 'a list -> 'a cycle_bien_chaîné
12 and        cycle_en_liste : 'a cycle_bien_chaîné -> 'a list
13 and          taille_cycle : 'a cycle_bien_chaîné -> int;;
```

Nous devrons définir deux fonctions d'insertion (devant et derrière l'élément courant), une fonction (`valeur_cycle`) qui renvoie l'élément courant, deux fonctions de déplacement dans le cycle (en avant ou en arrière), une fonction de suppression de l'élément courant, et bien sûr les conversions de listes circulaires en listes simples (à partir de l'élément courant) et inversement. C'est ce qui est fait ici.

Programme 3.2 La première partie du fichier `Cycles.ml` qui implémente les listes circulaires

```
type 'a cycle_bien_chaîné = Nil | Cellule of 'a cellule_de_cycle
 and 'a cellule_de_cycle =
    {   valeur                  : 'a;
        mutable lien_avant      : 'a cycle_bien_chaîné;
        mutable lien_arrière    : 'a cycle_bien_chaîné  };;

let avance_cycle = function Nil -> Nil | Cellule c -> c.lien_avant;;
let recule_cycle = function Nil -> Nil | Cellule c -> c.lien_arrière;;
let valeur_cycle = function Nil -> raise Cycle_Vide | Cellule c -> c.valeur;;

let lien_avant_sur but = function
      Nil -> raise Cycle_Vide
    | Cellule c -> c.lien_avant <- but;;


let lien_arrière_sur but = function
      Nil -> raise Cycle_Vide
    | Cellule c -> c.lien_arrière <- but;;

let taille_cycle = function
      Nil -> 0
    | cycle ->
        let rec compte n c = if c = cycle then n else compte (n+1) (avance_cycle c)
        in
        compte 1 (avance_cycle cycle);;

let est_singleton = function
      Nil -> false
    | c -> c = (avance_cycle c);;

let supprime_valeur_cycle = function
      Nil -> raise Cycle_Vide
    | c -> if (est_singleton c) then Nil
        else
        begin
            lien_avant_sur (avance_cycle c) (recule_cycle c);
            lien_arrière_sur (recule_cycle c) (avance_cycle c);
            avance_cycle c
        end;;
```

Programme 3.3 La seconde partie du fichier `Cycles.ml` qui implémente les listes circulaires

```
40 let insère_devant x cycle =
41     let c = Cellule {   valeur = x ;
42                         lien_avant = (avance_cycle cycle) ;
43                         lien_arrière = cycle }
44     in
45     if cycle = Nil then
46     begin
47         lien_avant_sur c c ;
48         lien_arrière_sur c c ;
49         c
50     end
51     else
52     begin
53         lien_avant_sur c cycle ;
54         lien_arrière_sur c (avance_cycle c) ;
55         c
56     end ;;
57
58 let insère_derrière x cycle =
59     let c = Cellule {   valeur = x ;
60                         lien_avant = cycle ;
61                         lien_arrière = (recule_cycle cycle) }
62     in
63     if cycle = Nil then
64     begin
65         lien_avant_sur c c ;
66         lien_arrière_sur c c ;
67         c
68     end
69     else
70     begin
71         lien_avant_sur c (recule_cycle c) ;
72         lien_arrière_sur c cycle ;
73         c
74     end ;;
75
76 let cycle_en_liste = function
77       Nil -> []
78     | c ->  let rec scan cycle accu =
79                 if cycle = c then (valeur_cycle c) :: accu
80                 else scan (recule_cycle cycle) ((valeur_cycle cycle) :: accu)
81             in scan (recule_cycle c) [] ;;
82
83 let rec liste_en_cycle = function
84       [] -> Nil
85     | a :: q -> insère_devant a (recule_cycle (liste_en_cycle q)) ;;
```

Chapitre 4

Files de priorité

4.1 Définition abstraite de la structure

Une file de priorité est une structure qui organise les éléments d'un ensemble muni d'une relation d'ordre, que nous noterons ici $\preceq$ (nous noterons la relation stricte $\prec$). Elle offre deux opérations fondamentales d'accès :

- l'insertion d'une nouvelle clé ;
- l'extraction de la clé qui réalise le minimum pour la relation $\preceq$.

On ajoutera souvent deux fonctions : l'une de création d'une file de priorité à partir d'une liste d'objets (c'est *a priori* une simple suite d'insertions), et l'autre qui procède à l'opération contraire, et rend la liste des objets contenus dans la structure.

Il existe plusieurs méthodes pour implémenter les files de priorité. On peut bien entendu commencer par essayer une simple liste. Si n est le nombre d'objets de la structure, l'insertion est alors en $O(1)$ et l'extraction du minimum est en $O(n)$.

On peut imaginer des structures plus efficaces, qui réalisent l'insertion et l'extraction en $O(\log n)$. C'est le cas de la structure en arbre bicolore, en tas, ou encore en file binomiale.

4.2 Une structure concrète : les tas

on dit aussi arbre tournoi parfait

la profondeur est le nombre d'arêtes parcourues pour aller de la racine à la feuille la plus basse

Des structures efficaces qui implémentent les files de priorité, la structure en tas est sans doute la plus simple à concevoir, la plus facile à mettre en œuvre, et, partant, la plus utilisée.

L'idée est simple : on part du fait qu'un arbre binaire de profondeur k peut loger, si on remplit tous les niveaux (sauf peut-être le dernier), entre 2^k et $2^{k+1} - 1$ objets. Inversement, un arbre bien rempli de n objets a une profondeur égale à $k = \lfloor \log_2 n \rfloor$.

On va donc tenter d'organiser les données en un arbre toujours bien rempli (on dit *parfait*), avec, pour retrouver facilement la clé minimale, une condition supplémentaire qui définit les arbres tournois et qui se traduit ainsi :

tout nœud de l'arbre a une valeur inférieure à celles de ses nœuds-fils

Toute la difficulté consiste à maintenir cette structure en autorisant les deux opérations souhaitées et en vérifiant toujours la condition imposée.

on parle de percolation

Lors d'une insertion, on ajoutera la nouvelle clé tout en bas de l'arbre et on la fera remonter, en l'échangeant avec son père, si celui-ci est supérieur, jusqu'à atteindre (peut-être) la racine de l'arbre (si on avait inséré une clé plus petite que toutes les autres). On trouvera, en figure 4.2, ce qui arrive quand on ajoute la clé 0 à l'arbre donné en exemple figure 4.1, ce qui correspond justement au cas le pire, où l'on remonte jusqu'à la racine.

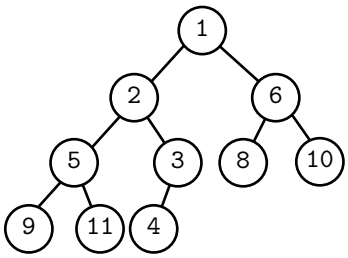


Figure 4.1: Un arbre tournoi parfait, *i.e.* un tas

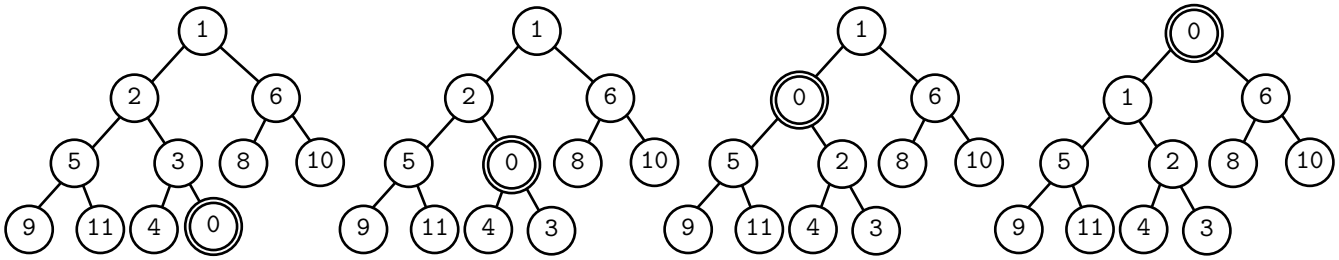


Figure 4.2: Insertion d'une nouvelle clé : la clé 0

Le minimum de la structure est toujours à la racine. On le trouve donc sans difficulté. Une fois l'arbre étêté, il faut faire remonter de ses deux fils le plus petit, créant ainsi un nouveau trou. On recommence alors, faisant à nouveau remonter le plus petit des deux fils éventuels, etc.

Le problème est qu'on n'est pas assuré que l'arbre final soit encore parfait, c'est-à-dire "bien" rempli. Dans l'exemple de la figure 4.3 où tout semble bien se passer, on s'aperçoit de notre erreur dans le cas où les clés 2 et 6 de l'arbre initial auraient été interverties : l'arbre final n'aurait plus été parfait.

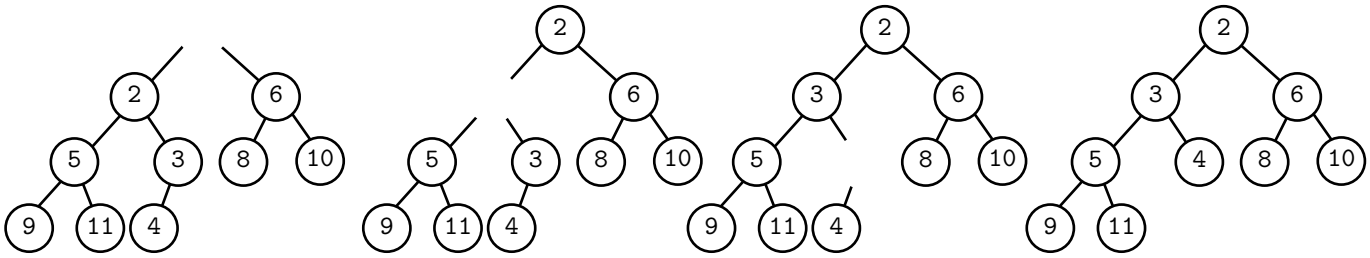


Figure 4.3: L'extraction de la clé minimale : version incorrecte

Une solution consiste à tout d'abord faire remonter à la racine l'élément le plus en bas à droite de l'arbre, c'est-à-dire en quelque sorte à créer le trou là où il ne pose pas problème : tout en bas. Ensuite seulement, on procède à la percolation, faisant redescendre à sa place la nouvelle racine. C'est ce qui est fait dans la figure suivante, la figure 4.4.

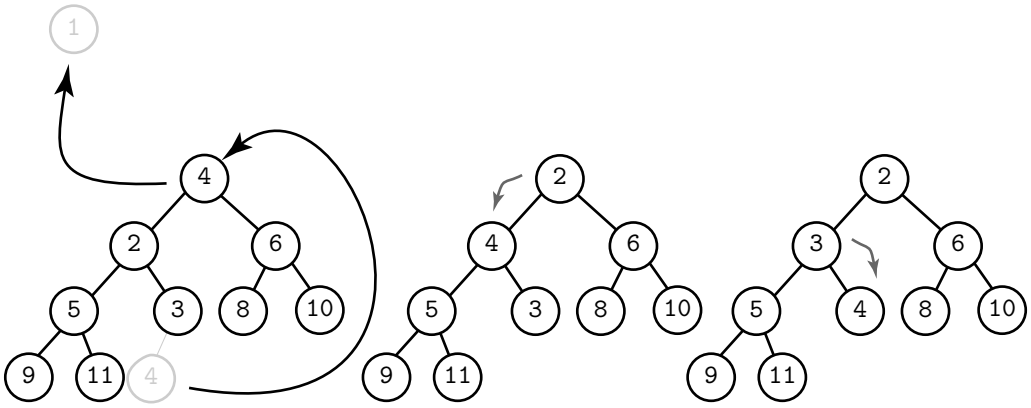


Figure 4.4: L'extraction de la clé minimale : version finale

4.3 Implémentation en Caml

Un autre intérêt de la structure de tas est qu'elle se prête particulièrement bien à une implémentation informatique simple. Nul besoin de créer un nouveau type `arbre` particulier, un simple tableau suffit à représenter un arbre parfait. En effet, si on numérote les nœuds d'un arbre parfait dans l'ordre dit militaire :

le plus âgé dans le grade le plus élevé

1. on numérote la racine de l'arbre par l'indice 0 ;
2. on numérote les nœuds de profondeur 1, de la gauche vers la droite ;
3. on numérote les nœuds de profondeur 2, de la gauche vers la droite ;
4. etc.

on vérifie que si i numérote un nœud, son père est numéroté $(i-1)/2$ (division entière, j'utilise la notation de Caml), et ses deux fils $2i+1$ et $2i+2$.

On représente donc un arbre parfait de taille n par un vecteur de même taille, et on utilisera la remarque précédente pour simuler le parcours de l'arbre.

Il est temps de donner l'interface `Tas.mli` (voir page suivante).

Un tas est un quintuplet `(t,donne_taille,incr_taille,<,taille_max)`. `t` est le vecteur des données, `donne_taille` est une fonction sans argument qui renvoie la taille courante du tas, `incr_taille` ajoute son argument à la taille actuelle, `<` désigne la relation d'ordre $\prec$, et `taille_max` est la taille-mémoire qu'on a réservée pour le tableau, et que ne doit pas dépasser la taile actuelle, sauf à déclencher l'exception `Tas_Plein`.

Nous avons ici utilisé une autre méthode classique de programmation Caml : nous définissons dans le programme `Tas.ml` quelques fonctions utilitaires comme par exemple `échange_dans_vect`, que nous ne déclarons pas dans `Tas.mli`. Après avoir chargé le module en mémoire grâce aux habituels `load_object "Tas" ;;` et `#open "Tas" ;;`, notre monde Caml ne contiendra que les définitions trouvées dans `Tas.zi`, c'est-à-dire ce qui a été généré durant la compilation de `Tas.mli`. En revanche les fonctions utiles, déclarées dans l'interface, restent disponibles pour l'utilisateur, comme par exemple `extrait_minimum`.

Programme 4.1 L'interface `Tas.mli`

```
1 type 'a structure_de_tas ;;
2
3 exception Tas_Plein ;;
4 exception Tas_Vide ;;
5
6 value percolation   : 'a structure_de_tas -> int -> unit
7 and insère_dans_tas : 'a structure_de_tas -> 'a -> unit
8 and extrait_minimum : 'a structure_de_tas -> 'a
9 and tas_en_liste    : 'a structure_de_tas -> 'a list
10 and tas_en_vecteur  : 'a structure_de_tas -> 'a vect
11 and liste_en_tas    : ('a -> 'a -> bool) -> 'a list -> int -> 'a structure_de_tas ;;
```

Nous aurons besoin, pour implémenter l'algorithme de Dijkstra sur les graphes, d'un accès à la procédure `remonte_dans_tas`. Nous avons choisi de déclarer publique une fonction `percolation` qui est simplement ici un synonyme de cette procédure. D'autres structures de données nous auraient conduits à une autre implémentation.

Voici quelques remarques sur la programmation des différentes fonctions de `Tas.ml` :

`échange_dans_vect` réalise simplement l'échange de deux éléments d'un vecteur quelconque ;

`descend_dans_tas` fait descendre à sa place dans le tas un élément donné, comme il est requis pour l'extraction. Il faut simplement faire attention à ne jamais aller au-delà de la taille du tas, `n`, qu'on récupère en ligne 25 ;

`remonte_dans_tas` fait au contraire remonter un élément, comme il est requis pour l'insertion. Il n'y a pas de difficulté particulière, et la programmation est plus simple que dans le cas précédent où on doit choisir la direction de la descente ;

`insère_dans_tas` doit seulement faire attention à la gestion de la taille du tas, et déclencher en cas de besoin l'exception `Tas_Plein` ;

`extrait_minimum` réalise l'extraction comme on l'a expliquée plus haut, et déclenche l'exception `Tas_Vide` si on l'appelle sur un tas initialement vide ;

`tas_en_liste` rend la liste des éléments du tas. Elle utilise une routine récursive locale toute simple `épuise` pour balayer le tableau ;

`tas_en_vecteur` rend un vecteur des éléments du tas ;

`liste_en_tas` crée l'arbre initial. On prendra garde, en particulier à cause de l'habituel problème du typage de Caml, à fournir une liste non vide d'éléments à ranger dans le tas. Si on tient vraiment à un tas initialement vide, on utilisera une liste comportant un élément unique pour fournir le type du tas, et on appellera immédiatement la procédure d'extraction. Bien sûr, il faut aussi fournir la relation d'ordre et la taille maximale à ne pas dépasser du tas. La procédure locale `copie_liste` réalise simplement la copie des éléments dans le tableau : `copie_liste i l` copie dans les éléments t_i, t_{i+1}, ..., les éléments de la liste `l`. Reste à transformer l'arbre que représente ce tableau en arbre tournoi : nous n'avons pour l'instant pas pris en considération la relation d'ordre entre les éléments. C'est le but de la ligne 101, mais cela mérite une preuve, qui est donnée ci-après.

on pourrait gérer une allocation de la mémoire comme on l'a fait pour les piles pour éviter ce problème

Programme 4.2 Début de `Tas.ml`, une gestion par les tas des files de priorité

```
(*=========================================================================*)
(*                                                                         *)
(*                     gestion de files de priorité                        *)
(*                  utilisation d'arbres tournois parfaits                 *)
(*                            ou tas   ou heaps                            *)
(*                                                                         *)
(*=========================================================================*)

(* définition du type *)
type 'a structure_de_tas =
    Tas of ('a vect * (unit -> int) * (int -> unit) * ('a -> 'a -> bool) * int);;

exception Tas_Plein;;
exception Tas_Vide;;

(*** PARTIE PRIVÉE :  ces fonctions ne sont pas déclarées dans Tas.mli,   *****)
(***                  et restent donc inaccessibles à l'utilisateur       *****)

let échange_dans_vect t i j =
    let temp = t.(i) in t.(i) <- t.(j); t.(j) <- temp;;

(* descend_dans_tas t i fait descendre à sa place dans le tas t *)
(* l'élément d'indice i                                         *)
let rec descend_dans_tas (Tas(tas,sa_taille,_,inf,_) as t) i =
    let n = sa_taille ()
    in                                          (* calcul du fils gauche *)
    let fils = 2*i + 1
    in                               (* calcul du plus petit des deux fils *)
    let fils = if (fils < n-1) & (inf tas.(fils+1) tas.(fils)) then fils+1 else fils
    in       (* descente de ce côté si le fils en question est plus petit *)
    if fils < n & inf tas.(fils) tas.(i) then
        begin
            échange_dans_vect tas i fils;
            descend_dans_tas t fils
        end;;

let rec remonte_dans_tas (Tas(tas,_,_,inf,_) as t) i =
    if i > 0 & inf tas.(i) tas.((i-1)/2) then
    begin
        échange_dans_vect tas i ((i-1)/2);
        remonte_dans_tas t ((i-1)/2)
    end;;

(********* FIN DE LA PARTIE PRIVÉE ***********************************************)
```

Programme 4.3 Fin de `Tas.ml`

```
45 (********* PARTIE PUBLIQUE :    ces fonctions sont déclarées dans Tas.mli,  *****)
46 (*********                      et sont disponibles pour l'utilisateur      *****)
47
48 let percolation = remonte_dans_tas ;;
49
50 let insère_dans_tas (Tas(tas,sa_taille,incr_taille,inf,nMAX) as t) x =
51     let n = sa_taille ()
52     in
53     if n = nMAX then raise Tas_Plein ;
54     tas.(n) <- x ;
55     incr_taille 1 ;
56     remonte_dans_tas t n ;;
57
58 let extrait_minimum (Tas(tas,sa_taille,incr_taille,_,_) as t) =
59     let n = sa_taille ()
60     in
61     if n = 0 then raise Tas_Vide ;
62     let mini = tas.(0)
63     in
64     begin
65         échange_dans_vect tas 0 (n-1) ;
66         incr_taille (-1) ;
67         descend_dans_tas t 0 ;
68         mini
69     end ;;
70
71 let tas_en_liste (Tas(tas,sa_taille,_,_,_) as t) =
72     let n = sa_taille ()
73     in
74     let rec épuise nb l = if nb = 0 then l else épuise (nb-1) (tas.(nb-1) :: l)
75     in épuise n [] ;;
76
77 let tas_en_vecteur t = vect_of_list (tas_en_liste t) ;;
78
79 (* on fournit la relation d'ordre <, une liste NON VIDE d'objets,   *)
80 (* et un majorant de la sa_taille du tas                            *)
81 (* on rend une structure de tas                                     *)
82
83 let liste_en_tas inf (a::q as l) nMAX =
84     let n = list_length l
85     in
86     let sa_taille = ref n
87     in
88     let tas = make_vect nMAX a
89     in
90     let rec copie_liste i = function
91           [] -> ()
92         | a::q -> tas.(i) <- a ; copie_liste (i+1) q
93     in
94     copie_liste 0 l ;
95     let t = Tas (   tas,
96                     (function () -> !sa_taille),
97                     (function delta -> sa_taille := !sa_taille + delta),
98                     inf,
99                     nMAX    )
100    in
101    for i = (n/2 - 1) downto 0 do descend_dans_tas t i done ;
102    t ;;
```

4.4 Preuve et évaluation de l'algorithme

Donnons maintenant une preuve de l'algorithme de création du tas (la ligne 101), et expliquons en particulier les bornes de variation de l'indice.

✧ Rappelons tout d'abord précisément la ligne en question :

```
for i = (n/2 - 1) downto 0 do descend_dans_tas t i done ;
```

Bien entendu, nous supposerons que `descend_dans_tas` fait ce qu'on attend de lui. Encore faut-il pour cela s'entendre sur sa définition précise.

`descend_dans_tas t i` demande que `i` soit le numéro d'un nœud tel que ses deux sous-arbres (éventuellement vides) gauche et droit soient déjà des tas ; alors il fait descendre le nœud à sa place, et le remplace par le minimum de telle sorte qu'après son exécution tout l'arbre de racine le nœud `i` soit lui-même un tas.

un tas *est un* arbre parfait, *c'est-à-dire un arbre bien rempli*

Ceci étant entendu, nous allons définir un invariant de boucle qui permettra de finir notre preuve. Cet invariant est le suivant : tout sous-arbre dont la racine est un nœud d'indice strictement supérieur à `i` est un tas.

Vérifions l'invariant à l'entrée de la boucle.

toujours une division entière à la *Caml*

Il s'agit de prouver que `n/2` est l'indice de la première (de gauche à droite) feuille de l'arbre. En effet les arbres réduits à un nœud sont nécessairement déjà des tas. Or on a déjà dit qu'un arbre de taille n a une profondeur égale à $k = \lfloor \log_2 n \rfloor$. Or, puisque nos divisions sont des divisions entières,

$$n/2 = 2^{\lfloor \log_2 n \rfloor - 1} = 2^{k-1},$$

et ainsi le plus gros arbre complet (son dernier niveau de profondeur est complètement rempli) strictement inclus dans notre arbre est de taille exactement égale à $n/2$. Donc $n/2$ est effectivement l'indice de la première feuille de notre tas, puisque notre numération commence à 0.

Le plus dur est fait.

conservé ou préservé ?

En effet, d'après ce que nous avons dit de `descend_dans_tas`, l'invariant de boucle est bel et bien conservé durant la boucle.

Et ceci finit la preuve, car en sortie de boucle on aura justement assuré que 0 est l'indice de la racine d'un tas, or c'est par construction la racine de notre arbre entier.

✦

Intéressons nous maintenant aux coûts des différentes fonctions écrites.

Soit n la taille du tas. Sa profondeur est alors $k = \lfloor \log_2 n \rfloor$, comme nous l'avons déjà dit à maintes reprises. Ainsi le coût d'une insertion est $O(k) = O(\log n)$ car il y aura au plus $k+1$ montées dans l'arbre à effectuer. De même le coût de l'extraction du minimum est $O(k) = O(\log n)$, ce n'est d'ailleurs pas l'extraction proprement dit qui coûte, mais la percolation nécessaire pour réorganiser le tas, et qui se traduit par au plus k descentes dans l'arbre.

Reste le problème de la création du tas à partir d'une liste de n objets, c'est-à-dire toujours et encore le cas de la ligne 101.

`descend_dans_tas t i`, quand $0 \leq i \leq n$, coûte au plus

$$k_i = \left\lfloor \log_2 \frac{n}{i+1} \right\rfloor,$$

car c'est la profondeur du sous-arbre du tas de racine `i`. Démontrons le :

✧ Fixons $n \geq 1$ et notons $k = \lfloor \log_2 n \rfloor$ la profondeur du tas. Posons, pour $i \geq 0$, $\varphi(i) = \left\lfloor \log_2 \frac{n}{i+1} \right\rfloor$.

Le dernier niveau du tas contient les feuilles à profondeur k, qui sont numérotées $2^k - 1$, $2^k, \ldots, n-1$. Or si $2^k - 1 \leq i \leq n-1$, alors $2^k \leq i \leq n$ et $-\log_2 n \leq -\log_2 i \leq -k$, d'où, d'après la définition de k, $\varphi(i) = 0$.

La propriété est donc vérifiée pour le dernier niveau, c'est-à-dire quand $i \geq 2^k - 1$.
Supposons la propriété vérifiée pour tous les nœuds de profondeur au moins égale à un certain j, où $1 \leq j \leq k$. Soit i le numéro d'un nœud à profondeur $j-1$, vérifions le résultat pour i, ce qui enclenchera la récurrence.
L'éventuel fils gauche de i porte le numéro $2i+1$, comme on l'a déjà dit.
De deux choses l'une : ou bien $2i+1 \geq n$, et c'est dire que i est une feuille ; ou bien la profondeur de l'arbre de racine i vaut $1 + \varphi(2i+1)$, car la propriété a été démontrée pour le fils gauche, qui est plus profond, et aussi car on est sûr que le sous-arbre gauche est au moins aussi profond que le sous-arbre droit, puisque l'arbre est supposé parfait.
Dans le premier cas, on trouve

$$\log_2 \frac{n}{i+1} \leq \log_2 \frac{n}{(n+1)/2} = 1 + \log_2 n - \log_2(n+1) < 1,$$

donc $\varphi(i) = 0$ et c'est gagné.
Dans le second cas, on a :

$$1 + \varphi(2i+1) = 1 + \left\lfloor \log_2 \frac{n}{2i+1+1} \right\rfloor = 1 + \left\lfloor \log_2 \frac{n}{i+1} - 1 \right\rfloor = \left\lfloor \log_2 \frac{n}{i+1} \right\rfloor = \varphi(i),$$

et ça marche encore.
◆

Le coût de la création du tas est donc

$$c(n) = \sum_{i=0}^{n/2-1} k_i = \sum_{i=1}^{n/2} \left\lfloor \log_2 \frac{n}{i} \right\rfloor = \sum_{i=1}^{+\infty} \left\lfloor \log_2 \frac{n}{i} \right\rfloor,$$

puisque les termes qui correspondent à $i > n/2$ sont tous nuls. Une étude superficielle donnerait à penser que $c(n) = O(n \log n)$. Il n'en est rien, de fait : $c(n) = O(n)$, ce que nous montrons maintenant.

✧ On va regrouper les termes de la somme en paquets liés aux puissances de 2 :

$$c(n) = \sum_{d=0}^{+\infty} \sum_{i=2^d}^{2^{d+1}-1} \left\lfloor \log_2 \frac{n}{i} \right\rfloor = \sum_{d=0}^{+\infty} \sum_{i=2^d}^{2^{d+1}-1} \left\lfloor \log_2 n - d - \log_2 x_i \right\rfloor,$$

où x_i est un réel tel que $1 \leq x_i < 2$, donc $0 \leq \log_2 x_i < 1$. Notons qu'en réalité, la plage de variation de l'indice d est la suivante : $0 \leq d \leq k-1$, où on a toujours $k = \lfloor \log_2 n \rfloor$.
On dispose alors de l'encadrement :

$$\sum_{d=0}^{k-1} 2^d (k-d-1) \leq c(n) \leq \sum_{d=0}^{k-1} 2^d (k-d).$$

Ces sommes s'évaluent aisément, et on obtient l'encadrement final

$$2^k - k - 1 \leq c(n) \leq 2^{k+1} - k - 2,$$

ce qui achève notre démonstration car $2^k = O(n)$ et $k \sim \log_2 n$.
◆

Chapitre 5

Partitions d'un ensemble

5.1 Définition abstraite de la structure

Le problème qui nous intéresse ici a été baptisé par les Anglo-Saxons *union-find*, car $union \equiv$ fusionner et $find \equiv$ trouver. Il interviendra par exemple dans certains algorithmes sur les graphes.

On considère un ensemble fini A de n objets a_0, a_1, ..., a_{n-1}. On remarquera qu'on ne demande aucune relation d'ordre particulière sur ces objets.

on peut très bien parler de l'histoire *de la structure*

On souhaite qu'à l'initialisation de la structure, on dispose d'une partition de A constituée des singletons $\{a_i\}$. C'est-à-dire qu'au début du processus on considère la partition

$$\mathcal{P} = \Big\{\{a_0\},\{a_1\},\ldots,\{a_{n-1}\}\Big\}.$$

Dans la suite du processus, on demande à disposer d'au moins deux opérations fondamentales :

une fonction retourne, *une procédure* procède *?*

- une fonction `trouve_paquet` qui retourne, pour un objet a_i de A, un entier caractéristique de celle des parties de la partition qui le contient.
- une procédure `fusionne_paquets` qui prenne deux entiers identifiant des éléments de la partition comme arguments et qui procède à l'agglutination des deux parties spécifiées.

Ainsi, après quelques étapes, on pourra se trouver, dans le cas où $n = 8$, avec la partition suivante :

$$\mathcal{P} = \Big\{\{a_0,a_2\},\{a_1\},\{a_4\},\{a_3,a_6,a_7\},\{a_5\}\Big\}.$$

On n'impose pas la valeur que doit rendre la fonction `trouve_paquet` quand on lui fournit en argument l'objet a_3, en revanche on veut que seuls a_3, a_6 et a_7 rendent le même résultat.

Enfin, pour poursuivre cet exemple, après l'appel de `fusionne_paquets` avec comme arguments les résultats de `trouve_paquet` sur a_0 et a_5, la partition courante devient :

$$\mathcal{P} = \Big\{\{a_0,a_2,a_5\},\{a_1\},\{a_4\},\{a_3,a_6,a_7\}\Big\}.$$

On pourra, pour la commodité de l'utilisateur, définir une fonction `regroupe_en_paquet` qui opère la même fusion mais qui attend en argument deux objets à mettre ensemble plutôt que les numéros des paquets correspondants.

5.2 Un autre point de vue

On peut aussi parler en termes de relation et de classes d'équivalence.

euxième est dite s grossière que la mière, ou ore. . . moins fine

Rappelons que si $\mathcal{R}_1$ et $\mathcal{R}_2$ sont deux relations d'équivalence sur un même ensemble E, la relation $\mathcal{R}_1$ est dite plus fine que la relation $\mathcal{R}_2$ si on a

$$\forall (x, y) \in E^2, x \mathcal{R}_1 y \Longrightarrow x \mathcal{R}_2 y.$$

Ainsi, par exemple, sur $\mathbb{N}$, peut-on dire que la relation *"a même parité que"* est moins fine que la relation *"a même reste modulo 6 que"*.

Si $\mathcal{R}_1$ est plus fine que $\mathcal{R}_2$, on observera que les partitions de E qui correspondent aux ensembles quotients $E/\mathcal{R}_1$ et $E/\mathcal{R}_2$ vérifient

$$\forall x \in E, \dot{x} \subset \ddot{x},$$

où je note $\dot{x}$ la classe d'un objet x modulo $\mathcal{R}_1$ et $\ddot{x}$ la classe du même objet modulo $\mathcal{R}_2$.

On part de la relation d'équivalence la plus fine sur l'ensemble de nos objets, à savoir celle pour laquelle les classes sont réduites aux seuls singletons. `trouve_paquet p a` fournit un représentant entier `na` de la classe de `a`. Si `na` et `nb` sont des représentants entiers des classes de `a` et `b`, `fusionne_paquets p na nb` transforme la relation d'équivalence de départ en une relation un peu moins fine, la plus fine de toutes celles qui sont à la fois moins fines que la première et qui vérifient $a \mathcal{R} b$.

perdu ? ah mais non ! faudrait voir à suivre un minimum !

5.3 Interface proposée

On donne l'interface `Partitions.mli`, sans la définition du type `partition` qui découlera de l'implémentation choisie.

Programme 5.1 L'interface `Partitions.mli` (sauf la définition des types)

```
1 value liste_en_partition : 'a list -> ('a -> int) -> 'a partition
2 and fusionne_paquets : 'a partition -> int -> int -> unit
3 and trouve_paquet : 'a partition -> 'a -> int
4 and nbre_de_paquets : 'a partition -> int
5 and même_paquet : 'a partition -> 'a -> 'a -> bool
6 and regroupe_2_en_paquet : 'a partition -> 'a -> 'a -> unit
7 and regroupe_n_en_paquet : 'a partition -> 'a list -> unit ;;
```

En ligne 1, est déclarée `liste_en_partition` qui prend en arguments une liste d'objets et une fonction de numérotation des objets à laquelle on demande d'être une bijection de A sur $\{0, \ldots, n-1\}$, et qui renvoie la partition initiale, composée des n singletons.

La fonction de numérotation des *objets* qui vient d'être évoquée n'a rien à voir avec les entiers qui sont associés de façon caractéristique à chacun des éléments de la partition (les paquets, si l'on préfère), c'est-à-dire les représentants entiers des classes. Il s'agit simplement de disposer de façon canonique (autrement dit indépendante du type des objets) d'un moyen de sélectionner tel ou tel objet.

`(fusionne_paquets p i j)` fusionne les éléments numéros `i` et `j` de la partition `p`.

`(trouve_paquet p a)` renvoie le numéro du paquet (l'entier qui le caractérise) de la partition `p` qui contient l'objet `a`.

`(nbre_de_paquets p)` et `(même_paquet p a b)` précisent le nombre d'éléments de la partition `p` ou si les objets `a` et `b` sont dans le même paquet.

Enfin, `(regroupe_2_en_paquet p a b)`, comme annoncé plus haut, procède à la fusion des éléments de la partition qui contiennent les deux objets `a` et `b`.

De même `(regroupe_n_en_paquet p [a ; ... ; z])` fusionne les éléments de la partition qui contiennent chacun des éléments de la liste donnée en dernier argument.

5.4 Une structure concrète

On pourrait bien sûr représenter la partition sous forme d'un tableau t de n entiers, tel que t_i contienne le numéro de sa classe. `trouve_paquet` est alors immédiat !

En revanche `fusionne_paquets` nécessite un balayage complet du tableau, et tourne donc en $O(n)$.

Nous allons utiliser plutôt une représentation arborescente : on associe à la partition une forêt d'arbres, chaque classe étant représentée par un arbre. Au sein d'une classe, les nœuds de l'arbre possèdent un pointeur sur leur père, sauf la racine, bien sûr. De fait, la structure complète peut être implémentée par un tableau, où t_i vaut i si t_i est la racine d'un arbre de la partition et le numéro j de son père dans le cas contraire.

pour simplifier la présentation, les objets de la partition sont représentés par des entiers

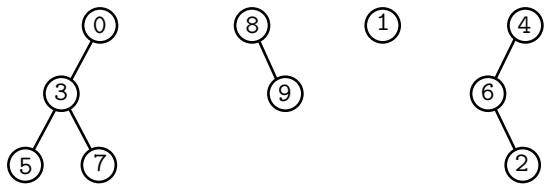


Figure 5.1: Une forêt associée à une partition

Par exemple, à la forêt représentée dans la figure ci-dessus, sont associés la partition $\{\{0,3,5,7\},\{1\},\{2,4,6\},\{8,9\}\}$ et le tableau suivant :

i	0	1	2	3	4	5	6	7	8	9
t_i	0	1	6	0	4	3	4	3	8	8

Dans la suite, pour éviter toute confusion, on désignera plutôt ce tableau par le nom `père`.

en général, il est conseillé d'utiliser des identificateurs très parlants et non ambigus

On aura bien entendu remarqué qu'il n'y a pas du tout unicité de la représentation. Notons d'autre part qu'il n'est pas nécessaire que les arbres de la forêt soient binaires. La figure suivante convient tout aussi bien que la précédente.

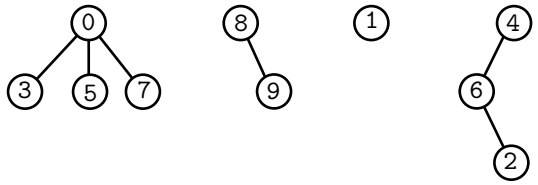


Figure 5.2: Une forêt moins profonde associée à la même partition

Il est bien clair que l'efficacité de la structure augmente quand la profondeur des arbres diminue. En effet, si la fusion de classes se fait maintenant en $O(1)$ quelles que soient les profondeurs des arbres, il faut, pour `trouve_paquet`, remonter d'un nœud d'un arbre jusqu'à sa racine.

raffiner pour équilibrer

La première idée qui permet d'équilibrer un peu les arbres est de raffiner la fusion. On va conserver dans un tableau `poids` des entiers représentant la profondeur de l'arbre qui se trouve sous chaque nœud. Au départ, il n'y a dans ce tableau que des 0 ; par la suite, lors d'une fusion de deux arbres, on s'arrangera pour limiter le poids global en créant un arbre dont la racine est celle de l'arbre *le plus lourd* à laquelle on accroche, comme arbre-fils, l'arbre

le plus léger. Bien sûr, on prendra garde à recalculer correctement le poids de la racine (ce qui ne doit être fait que si les deux arbres étaient de même poids).

Voici l'extrait correspondant de notre fichier final `Partitions.ml` :

```
if poids.(i) = poids.(j) then begin poids.(i) <- poids.(i) + 1 ; père.(j) <- i end
else if poids.(i) > poids.(j) then père.(j) <- i else père.(i) <- j ;;
```

Une autre façon d'arranger les choses, qui n'est pas contradictoire avec la précédente, est de remarquer que lors de la recherche de la classe d'un objet, quand on remonte vers la racine d'un arbre, on peut en profiter pour rediriger directement tous les nœuds visités vers la racine de l'arbre : ce sera autant de gagné pour la prochaine fois.

c'est le procédé dit "path compression" ou "compression des chemins"

Voici sur un exemple ce que cela donne : on cherche la classe de 20 ; la figure suivante montre l'arbre avant et après l'appel de `trouve_paquet p 20`.

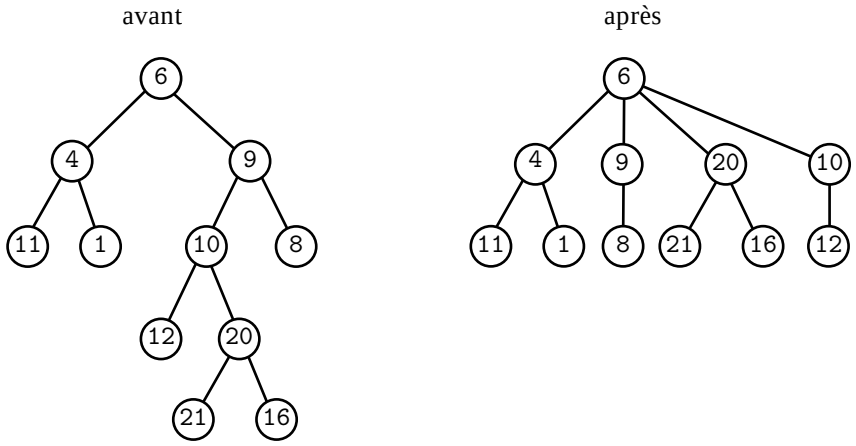


Figure 5.3: La compression des chemins

On écrira en Caml la recherche de la racine avec compression des chemins sous la forme

rappelons que l'on peut, en Caml, écrire `(... ; ...)` *au lieu de* `begin ... ; ... end`*, mais on tâchera de ne pas en abuser*

```
let rec monte_au_père i =   if père.(i) = i then i else monte_au_père père.(i)
in
let racine = monte_au_père x
in
let rec dirige_au_père i =
        if i = racine then ()
                      else ( dirige_au_père père.(i) ; père.(i) <- racine )
in dirige_au_père x ;
racine ;;
```

`dirige_au_père` est chargée de raccrocher tous les sommets au-dessus de `x` à la racine de l'arbre, comme nous l'avons indiqué.

Le lecteur attentif aura remarqué que les informations contenues dans le tableau `poids` ne représentent plus la profondeur effective des différents sous-arbres de la forêt, dès qu'on a utilisé la compression des chemins. C'est vrai. Il demeure que le critère — très simple — qui consiste à utiliser malgré tout ce tableau `poids` reste valable. Il serait finalement plus coûteux de maintenir les vraies profondeurs que de se contenter de l'information partielle dont on dispose avec ce tableau simplifié.

Dans la suite, nous utiliserons ces deux améliorations de l'algorithme.

5.5 Évaluation

on en doit l'étude exhaustive à Tarjan, puis Mehlhorn, qui a simplifié la présentation de la preuve de Tarjan

L'évaluation précise de la complexité de l'algorithme résultant de l'étude précédente peut être faite en détail, mais elle est vraiment très difficile. Nous nous contenterons ici de donner le résultat de cette étude.

On définit une fonction $A : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ par les relations de récurrence suivantes :

$$\begin{aligned} &\forall i \geq 1, && A(i,0) = 1; \\ &\forall j \geq 0, && A(0,j) = 2j; \\ &\forall (i,j), && A(i+1,j+1) = A(i, A(i+1,j)). \end{aligned}$$

Cette fonction est une variante de la célèbre fonction d'Ackermann. Elle a pour particularité de croître vraiment très rapidement.

On peut encore écrire facilement $A(i,j)$ pour $0 \leq i \leq 2$ et $0 \leq j \leq 6$:

j	0	1	2	3	4	5	6
0	0	2	4	6	8	10	12
1	1	2	4	8	16	32	64
2	1	2	4	16	65536	2^{65536}	$2^{2^{65536}}$

Mais ensuite on a par exemple

$$A(3,4) = 2^{\cdot^{\cdot^{\cdot^{2}}}} \quad ,$$

où il y a 65535 exposants 2 à la suite !

Rappel : $\lceil x \rceil$ désigne le plafond *de x, c'est-à-dire le plus petit entier supérieur à x*

Définissons maintenant un genre de fonction inverse de la fonction A. Soit α la fonction définie par :

$$\text{si } m \geq n, \text{ alors } \alpha(m,n) = \min\{p \geq 1 \ / \ A(p, 4\lceil m/n \rceil) > \log_2 n\}.$$

Évidemment, comme A croissait très vite, α est plutôt du style tortue qui lambine. Il est d'ailleurs facile d'établir que $\alpha(m,n) \leq \alpha(n,n)$ si $m \geq n$, puis que $\alpha(n,n) \leq 1$ tant que $n < 2^{16}$, et que $\alpha(n,n) \leq 2$ tant que $n < 2^{65536}$.

au diable la raison !

On retiendra que pour des valeurs raisonnables, $\alpha(m,n) \leq 2$. Bon, je vois des sceptiques : cherchez voir les entiers n tels que $\alpha(n,n) \geq 3$!

Tarjan démontre alors le

Théorème 1 (Tarjan) *Le coût de $n-1$ opérations de fusion (appels de `fusionne_paquets`) et de m opérations de recherche (appels de `trouve_paquet`) est $O(m\alpha(m,n))$.*

En général nous procéderons à $(n-1)$ opérations du genre *chercher les classes de deux objets puis les fusionner*, et nous pouvons donc dire qu'en général la complexité de notre algorithme est $O(n)$. (En réalité c'est $O(n\alpha(n,n))$, mais $\alpha(n,n)$ a peu de chances de dépasser quelques unités...)

5.6 Implémentation en Caml

listing ≡ listage

Venons-en enfin à notre implémentation en Caml, qu'on trouvera dans les deux listings suivants (l'interface d'abord, l'implémentation elle-même ensuite).

La description de l'algorithme et de ses détails majeurs d'implémentation a déjà été faite. Intéressons-nous simplement aux lignes 5–11 qui créent la partition. On fournit une liste d'objets et une fonction de numérotation de ces objets. `liste_en_partition` renvoie un objet de type `partition`, c'est-à-dire un quadruplet composé de la fonction de numérotation, du tableau des poids, de la taille et enfin du tableau `père`, toutes choses que nous avons décrites précédemment.

Programme 5.2 L'interface `Partitions.mli`

```
type 'a partition;;

value liste_en_partition    : 'a list -> ('a -> int) -> 'a partition
and fusionne_paquets        : 'a partition -> int -> int -> unit
and trouve_paquet           : 'a partition -> 'a -> int
and nbre_de_paquets         : 'a partition -> int
and même_paquet             : 'a partition -> 'a -> 'a -> bool
and regroupe_2_en_paquet    : 'a partition -> 'a -> 'a -> unit
and regroupe_n_en_paquet    : 'a partition -> 'a list -> unit;;
```

Programme 5.3 `Partitions.ml`

```
type 'a partition = Partition of ('a -> int) * (int vect) * int * (int vect);;

(* on fournit la liste d'objets, et une fonction qui permet de les numéroter *)
(* de 0 à n-1; on retourne la partition constituée des singletons *)
let liste_en_partition l numérote =
    let taille = list_length l
    in
    let poids = make_vect taille 0 and père = make_vect taille 0
    in
    for i = 0 to taille-1 do père.(i) <- i done;
    Partition(numérote,poids,taille,père);;

(* fusionne les deux parties numérotées i et j en une seule *)
let fusionne_paquets (Partition(_,poids,_,père) as p) i j =
    if i <> j then
    if poids.(i) = poids.(j) then begin poids.(i) <- poids.(i) + 1; père.(j) <- i
end
    else if poids.(i) > poids.(j) then père.(j) <- i else père.(i) <- j;;

(* renvoie le numéro de la partie qui contient l'objet a *)
let trouve_paquet (Partition(numérote,poids,_,père) as p) a =
    let x = numérote a
    in
    let rec monte_au_père i = if père.(i) = i then i else monte_au_père père.(i)
    in
    let racine = monte_au_père x
    in
    let rec dirige_au_père i =
         if i = racine then ()
                       else ( dirige_au_père père.(i); père.(i) <- racine )
    in dirige_au_père x; racine;;

(* renvoie le nombre de parties qui constituent la partition *)
let nbre_de_paquets = function Partition(_,_,n,père)
    -> let nb = ref 0
       in
       for i = 0 to n-1 do if i = père.(i) then nb := 1 + !nb done; !nb;;

(* dit si deux objets sont dans la même partie *)
let même_paquet p a b = (trouve_paquet p a) = (trouve_paquet p b);;

(* regroupe les paquets qui contiennent deux éléments donnés en un seul paquet *)
let regroupe_2_en_paquet p a b =
     fusionne_paquets p (trouve_paquet p a) (trouve_paquet p b);;

(* regroupe les paquets qui contiennent n éléments donnés en un seul paquet *)
let rec regroupe_n_en_paquet p = function
      [] -> ()
    | a :: [] -> ()
    | a :: b :: q -> regroupe_2_en_paquet p a b; regroupe_n_en_paquet p (a :: q);;
```

On aura noté la pratique adoptée ici : au lieu de rendre un n-uplet de fonctions d'accès et de modification à la structure, nous avons choisi de rendre un objet contenant les différentes variables caractéristiques de la structure, et de passer en argument à chacune des fonctions d'accès/modification la structure ainsi construite.
Il s'agit de deux points de vue assez différents. Le choix est affaire de goût (on aurait pu utiliser l'autre stratégie dans ce cas sans difficulté particulière, si l'on avait voulu). Remarquons que quel que soit le point de vue adopté, nous avons fait en sorte de limiter l'utilisation des variables globales : c'est là véritablement le critère essentiel.

Partie II

Quelques algorithmes sur les graphes

Chapitre 6

Généralités sur les graphes

6.1 Vocabulaire

Un graphe $G = (S, A, X, \varphi)$ est la donnée d'un ensemble fini non vide S dont les éléments sont appelés *sommets* du graphe, d'une partie A de $S \times S$, dont les éléments sont appelés *arêtes* du graphe et enfin d'une fonction $\varphi : A \to X$, où X est un ensemble quelconque, fini ou non, appelée *fonction de valuation* du graphe. Si $\alpha = (a, b) \in A$ est une arête du graphe, a et b sont deux sommets du graphe, et si $x = \varphi(\alpha)$, a s'appelle le *sommet de départ* de l'arête α, b le *sommet d'arrivée*, et x est l'*étiquette*, ou tout simplement la *valeur* de l'arête α.

le plus souvent $X = \mathbb{R}$

Dans la suite, nous noterons $a \xrightarrow{x} b$ pour exprimer à la fois que $(a, b) \in A$ et que $\varphi(a, b) = x$.

Notons que la plupart du temps on ne décrit pas un graphe de la façon précédente, mais avec un dessin, où les sommets sont représentés par leurs noms dans un cercle, les arêtes par des flèches, étiquetées par leurs valeurs.

Par exemple la figure suivante :

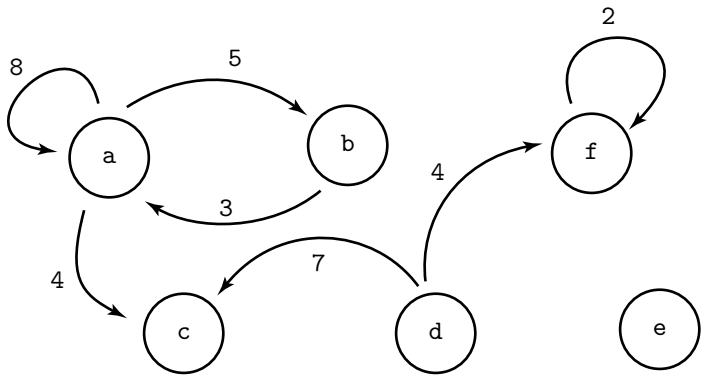


Figure 6.1: Un exemple de graphe

représente un graphe

$$G = (S, A, X, \varphi)$$

avec

$$\begin{aligned} S &= \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}, \\ A &= \{(\mathtt{a}, \mathtt{a}), (\mathtt{a}, \mathtt{b}), (\mathtt{a}, \mathtt{c}), (\mathtt{b}, \mathtt{a}), (\mathtt{d}, \mathtt{c}), (\mathtt{d}, \mathtt{f}), (\mathtt{f}, \mathtt{f})\}, \\ X &= \mathbb{R}, \end{aligned}$$

et où φ est représentée par le tableau suivant :

$u \to v$	a	b	c	d	e	f
a	8	5	4			
b	3					
c						
d			7			4
e						
f						2

Ainsi défini, un graphe est souvent dit *orienté* et *valué*.

Si $G = (S, A, X, \varphi)$ est un graphe orienté valué, on dira que $G' = (S, A)$, obtenu à partir de G en oubliant les étiquettes des arêtes, est le graphe non valué sous-jacent. Inversement, quand on nous fournira un graphe non valué $G' = (S, A)$, nous utiliserons sa représentation par le graphe valué $G = (S, A, \{\ \textit{vrai},\ \textit{faux}\}, \varphi)$, où $\varphi(\alpha)$ est définie constante égale à *vrai* pour toute arête $\alpha \in A$.

cette convention sera utilisée dans toute la suite

Le graphe $G = (S, A, X, \varphi)$ est dit symétrique si l'on a :

$$\forall (a, b) \in A, (b, a) \in A \text{ et } \varphi(b, a) = \varphi(a, b).$$

Notons que la structure de graphe orienté symétrique est isomorphe à celle de graphe non orienté et c'est pourquoi nous représenterons justement les graphes non orientés par des graphes symétriques.

imagine la définition rigoureuse d'un graphe non orienté

Enfin, il est souvent d'usage de prolonger la définition de φ à $S \times S$ tout entier. Pour cela nous poserons pour tout couple (a, b) qui n'est pas une arête $\varphi(a, b) =$ `nil_du_type`, où `nil_du_type` vaudra en général $+\infty$ si $X = \mathbb{R}$ et *faux* si $X = \{\ \textit{vrai},\ \textit{faux}\}$.

en toute rigueur, il faudrait prolonger $\mathbb{R}$ à $\mathbb{R} \cup \{+\infty\}$

En effet dans le cas où $X = \mathbb{R}$, l'étiquette représente souvent le coût, le poids, la longueur de l'arête, et donc l'absence d'arête correspond bien à un poids infini. Mais tel n'est pas toujours le cas : dans les problèmes dits *problèmes de flots*, l'étiquette représente au contraire la capacité du *tuyau* que représente l'arête, et il conviendra alors de définir `nil_du_type` $= 0$.

6.2 Une représentation avec des listes

La représentation informatique des graphes orientés valués que nous privilégierons est fondée sur l'utilisation de listes.

On représentera le graphe par une liste de couples de la forme (*nœud*, *liste d'adjacence*) où *nœud* décrit l'ensemble des sommets du graphe (même s'ils sont isolés, comme dans le cas du sommet `e` de notre exemple ci-dessus), et où la *liste d'adjacence* associée est une liste (éventuellement vide, donc) de couples de la forme (*voisin*, *poids*) telle que (*nœud*, *voisin*) décrive A et, bien sûr, *poids* $= \varphi($ *nœud*, *voisin*$)$.

une liste de couples d'un sommet et d'une liste de couples de ...

Dans notre exemple ci-dessus, on obtient la représentation suivante du graphe :

en une pseudo-syntaxe à la Caml

```
graphe = [
           ( a , [ (a,8) ; (b,5) ; (c,4) ] )   ;
           ( b , [ (a,3) ] )   ;
           ( c , [ ] )   ;
           ( d , [ (c,7) ; (f,4) ] )   ;
           ( e , [ ] )   ;
           ( f , [ (f,2) ] )
         ]
```

Notons en particulier qu'on impose d'écrire des couples du genre `( e , [ ] )` sans quoi on ne pourrait reconstituer l'ensemble des sommets du graphe.

Remarquons au passage que le nombre n de sommets du graphe est toujours la taille de la liste qui représente le graphe.

On peut reprocher à cette représentation de ne pas montrer clairement quels sont les graphes symétriques. En revanche, elle se révèle fort pratique à l'usage, et surtout économique, au contraire de la représentation matricielle que nous évoquerons plus loin.

Donnons maintenant la réelle définition Caml de cette représentation.

Programme 6.1 Définition Caml de la structure de graphe

```
1 type sommet = Sommet of string
2  and 'a flèche == sommet * 'a
3  and 'a graphe == (sommet * ('a flèche) list) list ;;
```

On aura remarqué que les sommets seront représentés par des expressions du genre `Sommet("a")`, c'est volontaire, même si certains trouveront cela trop lourd : il s'agit de ne pas pouvoir mélanger les sommets avec les étiquettes ou autres...

6.3 Une représentation matricielle

Une autre solution pour représenter un graphe consiste à utiliser une liste des sommets (numérotés comme toujours en Caml de 0 à $n-1$) et une matrice carrée M d'ordre n (les indices (i,j) varient dans $\{0,\dots,n-1\}\times\{0,\dots,n-1\}$). On trouvera en ligne i et colonne j l'élément $m_{i,j} = \Phi(\varphi(\mathit{sommet}_i, \mathit{sommet}_j))$, où Φ est une transformation (qui sera souvent l'identité) de X en un ensemble éventuellement plus pratique. On notera que l'usage de `nil_du_type` qu'on a évoqué plus haut est ici incontournable : tous les éléments de la matrice doivent bien valoir quelque chose !

Par exemple, et avec tout simplement $\Phi = \mathrm{Id}$, on obtient la matrice suivante :

$$\begin{pmatrix} 8 & 5 & 4 & +\infty & +\infty & +\infty \\ 3 & +\infty & +\infty & +\infty & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty & +\infty & +\infty \\ +\infty & +\infty & 7 & +\infty & +\infty & 4 \\ +\infty & +\infty & +\infty & +\infty & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty & +\infty & 2 \end{pmatrix}.$$

On aura remarqué sur cet exemple le gaspillage auquel peut mener cette représentation.

6.4 Conversion d'une représentation à l'autre

On va donner maintenant les procédures Caml qui permettent de passer d'une représentation à l'autre.

Commençons par la fonction `matrice_d_incidence_et_sommets` qui prend en arguments le graphe défini par listes, la valeur de `nil_du_type` et la fonction Φ. Elle renvoie un couple constitué de la matrice cherchée et de la liste des noms des sommets.

La fonction inverse, `représentation_en_liste`, prend en arguments la matrice, la valeur de `nil_du_type`, et la liste des noms des sommets. Elle renvoie la liste qui représente le même graphe.

Programme 6.2 Conversion de la représentation en listes vers la représentation matricielle

```
(*====== matrice_d_incidence_et_sommets ========================================*)

(* renvoie un couple (m,s)                              *)
(*  où m est la matrice de terme générique m(i,j) égal  *)
(*  à l'image par la fonction transf                    *)
(*  de la valeur de l'arc (i->j) s'il existe,           *)
(*  ou à nil_du_type sinon                              *)
(*  et s est la liste des noms des sommets du graphe    *)

let matrice_d_incidence_et_sommets g nil_du_type transf =
    let n = list_length g
    in
    let m = make_matrix n n nil_du_type
    and s = let rec crée_liste_sommets = function
                   [] -> []
               | (Sommet(s),_) :: q -> s :: crée_liste_sommets q
            in crée_liste_sommets g
    in
    let rec ajoute_flèche i = function
          [] -> ()
        | (Sommet(but),val) :: qf
            -> m.(i).(index but s) <- transf val ;
               ajoute_flèche i qf
    and crée_matrice =  function
              [] -> ()
            | (Sommet(a),f) :: q
                -> ajoute_flèche (index a s) f ;
                   crée_matrice q
    in
    crée_matrice g ;
    (m,s) ;;
```

Programme 6.3 Conversion de la représentation matricielle vers la représentation en listes

```
(*====== opération inverse : donner la liste à partir de la matrice ==============*)

let représentation_en_liste m nil_du_type sommets =
    let n = list_length sommets
    and sv = vect_of_list sommets
    in
    let rec crée_liste_de i j petit_accu =
        if j = n then petit_accu
        else crée_liste_de  i (j+1)
                            (   if m.(i).(j) = nil_du_type
                                then petit_accu
                                else ((Sommet(sv.(j)),m.(i).(j)) :: petit_accu)  )
    and crée_liste i accu =
        if i = n then accu
        else crée_liste (i+1) ((Sommet(sv.(i)),(crée_liste_de i 0 [])) :: accu)
    in
    crée_liste 0 [] ;;
```

Chapitre 7

Les algorithmes de Floyd et de Warshall

7.1 Le problème des plus courts chemins

Considérons le réseau ferré de la SNCF. On peut le représenter par un graphe dont les sommets sont les gares de France, les arêtes les lignes ferroviaires, et les étiquettes les distances kilométriques. Notons que, par définition même, le graphe est symétrique.

trouver la distance et les gares intermédiaires

Si il est clair que pour aller de Paris à Rouen on utilisera la ligne directe, il est en revanche moins évident de trouver l'itinéraire le plus court entre Metzeral (en Alsace) et Luzy (dans le Morvan). On cherche plus généralement un moyen de calculer l'itinéraire le plus court d'une gare quelconque à une autre. Je sais : les méchants diront que j'oublie le problème essentiel, à savoir les correspondances. Mais à chaque jour suffit sa peine.

Explicitons de façon plus mathématique le problème, en utilisant (bien sûr !) les notations introduites dans le chapitre précédent.

Considérons donc un graphe $G = (S, A, X, \varphi)$, où X est muni d'une relation d'ordre total $\preceq$ et d'une loi de composition interne $+$ qui en font un groupe ordonné. Dans la suite, nous utiliserons le langage correspondant au cas courant $X = \mathbb{R}$, et nous appellerons par exemple l'élément neutre 0, nous dirons que x est positif si $0 \preceq x$, etc.

nos chemins sont donc orientés, y faire bien attention pour un graphe non symétrique

On appelle *chemin* d'un sommet $a \in S$ à un sommet $b \in S$ toute suite finie $(a_0, a_1, \ldots, a_p)$ de sommets de S telle que $a_0 = a$, $a_p = b$ et, pour tout i tel que $1 \leq i \leq p$, le couple (a_{i-1}, a_i) soit élément de A, c'est-à-dire soit une arête du graphe.

On dira que le graphe G est *connexe* si pour tout couple (a, b) de sommets distincts il existe un chemin de a à b.

Si $(a_0 = a, a_1, \ldots, a_p = b)$ est un chemin de a à b, sa *longueur* (ou son poids, tout dépend de l'interprétation qu'on choisit), est définie par

$$\ell(a_0, \ldots, a_p) = \sum_{i=1}^{p} \varphi(a_{i-1}, a_i).$$

Nous allons démontrer le résultat suivant :

Théorème 2 *Si G est un graphe connexe, et si pour toute arête α on a $0 \preceq \varphi(\alpha)$, alors, pour tout couple fixé (a, b) de sommets distincts de G, l'ensemble des longueurs des différents chemins de a à b admet un plus petit élément pour l'ordre défini par $\preceq$.*

Un chemin qui réalise cette longueur minimale s'appellera un *plus court chemin* de a à b.

✧ Fixons a et b. Notons $\mathcal{C}$ l'ensemble des chemins de a à b, et $\mathcal{L}$ l'ensemble de leurs longueurs.
Comme G est un graphe connexe, $\mathcal{C}$ et donc aussi $\mathcal{L}$ sont non vides.

Soit $\ell_0 = \ell(c_0)$ la longueur d'un chemin particulier c_0 de a à b.
Nous allons montrer que l'ensemble $\mathcal{L}_0$ des longueurs des chemins de a à b qui sont plus petites que ℓ_0 ($\mathcal{L}_0 = \{\ell \in \mathcal{L} \,/\, \ell \preceq \ell_0\}$) est *fini*. Le résultera en découlera aussitôt.
mais pas forcément l'ensemble des chemins correspondants
Soit A^* l'ensemble des arêtes de longueur nulle du graphe.
Posons $m = \inf\{\ell(\alpha), \alpha \in A \setminus A^*\}$, m est la plus petite longueur non nulle des arêtes du graphe.
Si donc on considère un chemin $c = (a_0 = a, a_1, \ldots, a_p = b)$ de a à b dont la longueur est dans $\mathcal{L}_0$, on peut lui associer la suite $\hat{c}$ de ses arêtes qui ne sont pas dans A^*. Alors $\ell(c) = \sum_{\alpha\in\hat{c}} \varphi(\alpha) \succeq m \times |\hat{c}|$, où $|\hat{c}|$ désigne le nombre d'arêtes dans $\hat{c}$.
Posons $k_0 = \lfloor \ell_0/m \rfloor$.
Convenons d'appeler poids total d'un ensemble d'arêtes la somme de leurs longueurs.
$\mathcal{L}_0$ est alors inclus dans l'ensemble des poids totaux des suites d'au plus k_0 éléments de $A \setminus A^*$. Comme ce dernier ensemble est fini, $\mathcal{L}_0$ aussi, et ma preuve itou.
◆

Notons que le résultat n'est pas assuré si on admet des arêtes de longueur négatives. Par exemple dans le cas du graphe de la figure suivante, il n'y a pas de plus court chemin de a à b. Il suffit pour s'en convaincre d'examiner les poids des chemins ab, $acab$, $acacab$, ...

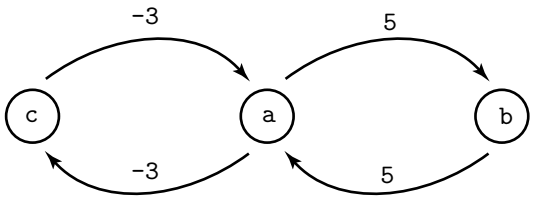


Figure 7.1: Un graphe sans plus court chemin

Notre problème est maintenant clairement défini. On considère un graphe G connexe et d'arêtes de longueurs toutes positives ou nulles. On veut pour tout couple de sommets distincts trouver la longueur d'un plus court chemin qui les relie et exhiber un tel plus court chemin.

Dans la suite les hypothèses requises (graphe connexe, longueur d'arêtes non négatives) sont supposées vérifiées.

7.2 L'algorithme de Floyd

L'algorithme de Floyd répond très exactement au problème que nous nous sommes fixé.
Floyd publie son algorithme en 1962

Nous supposerons en outre que $X \subset \mathbb{R}$.

Pour alléger les notations nous conviendrons de numéroter les n sommets du graphe, qui seront confondus, dans cette section, avec leurs numéros. Ainsi a-t-on $S = \{0, 1, \ldots, n-1\}$.

Soit M la matrice des longueurs des arêtes : $m_{i,j}$ est égal à $\varphi(i,j)$ si $(i,j) \in A$, et à $+\infty$ sinon (on étend comme d'habitude les définitions de $+$ et $\leq$ à $\overline{\mathbb{R}} = \mathbb{R} \cup \{+\infty\}$).

L'algorithme repose sur la remarque suivante, qui est de démonstration immédiate : si $(a_0, \ldots, a_i, \ldots, a_p)$ est un plus court chemin de a_0 à a_p, alors nécessairement $(a_0, \ldots, a_i)$ est un plus court chemin de a_0 à a_i et $(a_i, \ldots, a_p)$ est un plus court chemin de a_i à a_p.

En outre comme un chemin qui contient un cycle a une longueur au moins égale au chemin obtenu en supprimant ce cycle :

$$\ell(a_0, \ldots, a_i, \ldots, a_i, \ldots, a_p) \succeq \ell(a_0, \ldots, a_i, \ldots, a_p),$$

on peut se limiter aux plus courts chemins passant par des sommets deux à deux distincts.

Floyd propose de calculer la suite de matrices $M^{(-1)}$, $M^{(0)}$, $M^{(1)}$, $M^{(2)}$, ..., définie par $M^{(-1)} = M$, et, pour $k \geq 0$,

$$M_{i,j}^{(k)} = \min\{M_{i,j}^{(k-1)}, M_{i,k}^{(k-1)} + M_{k,j}^{(k-1)}\}.$$

Une récurrence montre sans difficulté aucune que, quand $k \geq 0$, $M_{i,j}^{(k)}$ est égal à la longueur du plus court chemin de i à j, quand on ne s'autorise — en fait de sommets intermédiaires — que les sommets $0, 1, \ldots, k$.

Il est alors clair d'après les remarques précédentes qu'après n calculs de matrices, $M^{(n-1)}$ est finalement la matrice des longueurs des plus courts chemins.

Bien entendu, le calcul de chaque matrice se faisant en $O(n^2)$, l'algorithme de Floyd tourne en $O(n^3)$.

Il reste une question à laquelle il semble que nous n'avons pas répondu : quels sont les plus courts chemins? En fait, il suffit de remarquer que quand, dans le calcul de $M^{(k)}$ nous remplaçons $M_{i,j}^{(k-1)}$ par la somme $M_{i,k}^{(k-1)} + M_{k,j}^{(k-1)}$, c'est que nous savons que le plus court chemin (en tout cas jusqu'à nouvel ordre) passe par k. On retiendra donc dans une matrice supplémentaire, et au fur et à mesure du calcul des matrices $M^{(k)}$, le sommet par lequel il faut passer pour aller de i à j. Sachant qu'il faut passer par k, il suffira de considérer — récursivement — les éléments d'indices i et k d'une part et k et j d'autre part, pour reconstituer l'intégralité du plus court chemin de i à j.

7.3 Implémentation en Caml de $\mathbb{N} \cup \{+\infty\}$

réécris les instructions correspondant au cas réel

Nous devons tout d'abord définir en Caml les opérations qui font intervenir $+\infty$. Nous avons choisi de considérer des arêtes entières, et nous écrirons simplement :

Programme 7.1 L'ensemble $\mathbb{N} \cup \{+\infty\}$

```
1 (* on introduit un type correspondant à N U { +infini } *)
2 type entier = Infini | N of int ;;
3
4 let inférieur x y = match x,y with
5       Infini,Infini -> true
6     | Infini,N(_) -> false
7     | N(_),Infini -> true
8     | N(p),N(q) -> p <= q ;;
9
10 let strict_inférieur x y = match x,y with
11       Infini,Infini -> false
12     | Infini,N(_) -> false
13     | N(_),Infini -> true
14     | N(p),N(q) -> p < q ;;
15
16 let plus x y = match x,y with
17       Infini,Infini -> Infini
18     | Infini,N(_) -> Infini
19     | N(_),Infini -> Infini
20     | N(p),N(q) -> N(p + q) ;;
```

7.4 Implémentation en Caml de l'algorithme de Floyd

Nous écrivons en Caml le programme ci-dessous.

Programme 7.2 L'algorithme de Floyd

```
1 (* pour améliorer la lisibilité *)
2 type intermédiaire = Direct | Nœud of string;;
3
4 (* floyd g renvoie une paire de deux fonctions distance,chemin          *)
5 (* distance Sommet(a) Sommet(b) renvoie la plus courte distance de a à b *)
6 (* chemin Sommet(a) Sommet (b) décrit le chemin à utiliser              *)
7
8 let floyd g =
9     let n = list_length g
10    and m,sl = matrice_d_incidence_et_sommets g Infini (function x -> N(x))
11    in
12    let sv = vect_of_list sl
13        (***** a priori les chemins sont tous directs                    *)
14    and passage = make_matrix n n Direct
15    in
16        (***** le cœur de l'algorithme de Floyd : du O(n^3) à coup sûr   *)
17    for k = 0 to n-1 do
18        for i = 0 to n-1 do
19            for j = 0 to n-1 do
20                let raccourci = plus m.(i).(k) m.(k).(j)
21                in
22                if strict_inférieur raccourci m.(i).(j) then
23                begin
24                    m.(i).(j) <- raccourci;
25                    passage.(i).(j) <- Nœud (sv.(k))
26                end
27            done
28        done
29    done;
30    let rec distance (Sommet a) (Sommet b) = m.(index a sl).(index b sl)
31        (***** une petite récursion pour trouver le chemin de a à b      *)
32    and chemin (Sommet a) (Sommet b) =
33        match passage.(index a sl).(index b sl) with
34        Direct -> [(Sommet a); (Sommet b)]
35        | Nœud c -> (chemin (Sommet a)(Sommet c))
36                    @
37                    (tl (chemin (Sommet c)(Sommet b)))
38    in
39    (distance,chemin);;
```

il a fallu inventer un nouveau constructeur, `Nœud`, car `Sommet` n'est pas disponible pour un nouveau type

La ligne 2 permet de gérer la matrice `passage` qui servira à la reconstitution des plus courts chemins. Elle sera initialisée avec `Direct`, ce qui indique qu'on n'a pas encore trouvé de sommet intermédiaire pour réaliser un raccourci.

On aura noté aussi que plutôt qu'écrire n matrices différentes successives, on met simplement à jour les éléments d'une matrice unique. La matrice initiale est remplie en ligne 10 par des éléments de $\mathbb{N} \cup \{+\infty\}$ grâce à notre fonction `matrice_d_incidence_et_sommets` (vue au chapitre précédent) à laquelle on passe le graphe, la valeur `Infini`, et la fonction qui transforme un entier standard (`int`) en un `entier`.

En lignes 31–37 on trouvera la petite procédure récursive qui reconstitue les plus courts chemins, à l'aide de la matrice `passage` qui a été mise à jour en ligne 25.

Enfin, comme on en a vu des exemples analogues dans la première partie de ce poly, la procédure `floyd` renvoie deux *fonctions*, `distance` et `chemin`, qui, appliquées à deux sommets renvoient respectivement la longueur du plus court chemin qui les relie et un de ces plus courts chemins. Ainsi le calcul (en $O(n^3)$) réalisé dans l'algorithme de Floyd n'est-il effectué qu'une seule fois, après quoi on peut s'en servir à volonté.

Montrons sur un exemple comment utiliser notre programme.
On considère le graphe suivant : Quel est un plus court chemin de `a` à `c`?

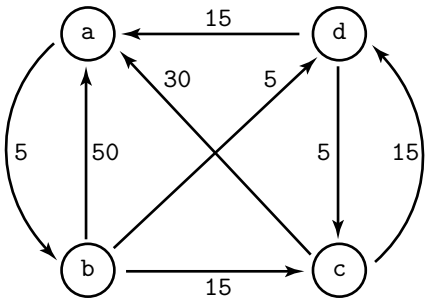


Figure 7.2: Un graphe exemple pour l'algorithme de Floyd

Caml nous répond :

Programme 7.3 La réponse de Caml sur notre exemple pour l'algorithme de Floyd

```
1 #let g = [  (Sommet "a") , [ (Sommet "b"),5 ] ;
2             (Sommet "b") , [ (Sommet "a"),50 ; (Sommet "c"),15 ; (Sommet "d"),5 ] ;
3             (Sommet "c") , [ (Sommet "a"),30 ; (Sommet "d"),15 ] ;
4             (Sommet "d") , [ (Sommet "a"),15 ; (Sommet "c"),5 ]     ] ;;
5 g : (sommet * (sommet * int) list) list = [Sommet "a", [Sommet "b", 5]; Sommet "b",
  [Sommet "a", 50; Sommet "c", 15; Sommet "d", 5]; Sommet "c", [Sommet "a", 30; Sommet
  "d", 15]; Sommet "d", [Sommet "a", 15; Sommet "c", 5]]
6
7 #let (distance,chemin) = floyd g ;;
8 chemin : sommet -> sommet -> sommet list = <fun>
9 distance : sommet -> sommet -> entier = <fun>
10
11 #distance (Sommet "a") (Sommet "c") ;;
12 - : entier = N 15
13
14 #chemin (Sommet "a") (Sommet "c") ;;
15 - : sommet list = [Sommet "a"; Sommet "b"; Sommet "d"; Sommet "c"]
16
17 #
```

7.5 L'algorithme de Warshall

L'utilisation que nous avons faite des entiers étendus ($\mathbb{N} \cup \{+\infty\}$) revient à dire que, quitte à considérer des longueurs infinies d'arêtes, notre graphe est toujours connexe.

Imaginons ce que donnerait l'algorithme de Floyd sur un graphe dont toute arête serait de longueur infinie (elle ne figure pas dans le graphe) ou de longueur nulle. La longueur d'un plus court chemin entre deux sommets sera alors ou nulle (les deux sommets sont reliés) ou infinie (les deux sommets sont dans deux composantes connexes disjointes). Le plus court chemin trouvé par l'algorithme serait alors simplement un exemple de chemin reliant les deux sommets, quand il existe.

pourquoi cette appellation ?

Finalement nous résolvons ainsi la question qu'on appelle communément *problème de la fermeture transitive du graphe*. Bien sûr, il ne sert à rien d'étiqueter par des entiers les arêtes. On utilisera un étiquetage par des booléens : `true` pour indiquer que l'arête existe, et `false` jouera le rôle jusqu'alors dévolu à l'infini. En outre, ceci permet de simplifier la récurrence

qui définit les matrices successives : on aura simplement ici, avec les notations précédentes,

$$M_{i,j}^{(k)} = M_{i,j}^{(k-1)} \text{ ou } \left(M_{i,k}^{(k-1)} \text{ et } M_{k,j}^{(k-1)} \right).$$

L'algorithme ainsi obtenu s'appelle l'algorithme de Warshall. *il date de 1962*
On déduit son implémentation en Caml de celle qu'on a écrite pour l'algorithme de Floyd.

Programme 7.4 L'algorithme de Warshall

```
 1 (*========================================================================*)
 2 (*                                                                        *)
 3 (*             recherche de la fermeture transitive du graphe             *)
 4 (*                          algorithme de Warshall                        *)
 5 (*                                                                        *)
 6 (*========================================================================*)
 7
 8 (* warshall g renvoie une paire de deux fonctions reliés,chemin           *)
 9 (* reliés est une fonction booléenne qui dit s'il existe un chemin de a vers b  *)
10 (* chemin Sommet(a) Sommet (b) décrit le chemin à utiliser               *)
11
12 let warshall g =
13     let n = list_length g
14     and m,sl = matrice_d_incidence_et_sommets g false (function x -> true)
15     in
16     let sv = vect_of_list sl
17     and passage = make_matrix n n Direct
18     in
19         (***** le cœur de l'algorithme de Warshall : du O(n^3) à coup sûr   *)
20     for k = 0 to n-1 do
21         for i = 0 to n-1 do
22             for j = 0 to n-1 do
23                 if m.(i).(k) & m.(k).(j) then
24                 begin
25                     m.(i).(j) <- true ;
26                     passage.(i).(j) <- Nœud (sv.(k))
27                 end
28             done
29         done
30     done ;
31     let rec reliés (Sommet a) (Sommet b) = m.(index a sl).(index b sl)
32         (***** une petite récursion pour trouver le chemin de a à b         *)
33     and chemin (Sommet a) (Sommet b) =
34         match passage.(index a sl).(index b sl) with
35         Direct -> [(Sommet a) ; (Sommet b)]
36         | Nœud c -> (chemin (Sommet a)(Sommet c))
37                     @
38                     (tl (chemin (Sommet c)(Sommet b)))
39     in
40     (reliés,chemin) ;;
```

On aura noté en ligne 14 la fonction `(function x -> true)` qui remplace toute étiquette d'une arête existante en l'étiquette `true`, et l'utilisation de `false` pour `nil_du_type`.

Chapitre 8

L'algorithme de Dijkstra

8.1 Utilité d'un nouvel algorithme

L'algorithme de Floyd, comme nous l'avons vu dans le chapitre précédent, permet de trouver une solution au problème des plus courts chemins dans un graphe de n sommets en $O(n^3)$. Le moins qu'on puisse dire c'est que $O(n^3)$ c'est trop : on évite habituellement comme la peste les algorithmes quadratiques (c'est-à-dire en $O(n^2)$), alors $O(n^3)$!!

les algorithmes sont comme les réalisateurs, ils tournent...

Nous allons décrire ici un algorithme qui tourne en $O(a \log n)$, où a est le nombre d'arêtes du graphe (supposé connexe). De fait, l'algorithme ne produit pas tout à fait le même résultat : on se fixe un sommet, appelé source, et on cherche les plus courtes distances de cette source à chacun des autres sommets du graphe, ainsi qu'un plus court chemin pour chacun.

Cet algorithme est dû à Dijkstra, il a été publié en 1959. Il fait partie de la grande famille des algorithmes baptisés par les Anglo-Saxons *greedy algorithms*, ce qu'on pourrait traduire par *algorithmes gloutons*.

mettons-nous en appétit

8.2 Description de l'algorithme de Dijkstra

tu vérifieras que sinon l'algorithme de Dijkstra peut sombrer dans une boucle infinie

L'algorithme de Dijkstra s'applique à un graphe dont toutes les arêtes sont étiquetées par des valeurs positives (ici encore nous prenons $X = \overline{\mathbb{R}} = \mathbb{R} \cup \{+\infty\}$), sans quoi le problème posé, nous l'avons vu, n'a pas de solution.

On se fixe donc un sommet source, et on va maintenir tout au long du déroulement de notre algorithme deux ensembles de sommets, que nous appellerons C et Δ.

le nôtre *? celui de Edsger Wybe D. !*

C est l'ensemble des sommets qui restent à visiter : au départ $C = S \setminus \{\text{source}\}$.

Δ est l'ensemble des sommets pour lesquels on connaît déjà leur plus petite distance à la source : au départ $\Delta = \{\text{source}\}$.

On aura toujours $S = C \cup \Delta$, de telle sorte qu'on s'arrêtera quand $C = \emptyset$.

Pour chaque sommet s dans Δ, on conservera dans un tableau `distances` sa plus courte distance à la source, et dans un tableau `passage` le sommet p qui le précède dans le plus court chemin choisi de la source à s : ce plus court chemin sera de la forme $(\text{source}, \ldots, p, s)$ — ce n'est pas la convention que nous avions adoptée pour Floyd. Il sera facile de récupérer un plus court chemin avec un (unique, cette fois) appel récursif, en remontant de prédécesseur en prédécesseur jusqu'à la source.

Comment allons-nous donc faire "grossir" Δ tout en respectant les conditions imposées?

On initialise le tableau `distances` par les étiquettes des arêtes de la source à chacun des sommets de $C = S \setminus \{\text{source}\}$, et de même `passage` par source, pour tout le monde. On peut interpréter cette initialisation de la façon suivante : jusqu'à ce qu'on trouve peut-être mieux, les informations dont nous disposons pour le moment font que les plus courts chemins observés sont simplement les arêtes qui relient la source à chacun des sommets.

glouton, mais pas goinfre

À chaque étape de l'algorithme, Δ va s'enrichir d'un nouveau sommet. Si donc le graphe

est connexe, ce que nous supposerons dans la suite, nous en aurons terminé en $n-1$ étapes (on verra tout à l'heure que la $(n-1)$-ème et dernière étape peut être ignorée, car on n'y fera strictement rien).

Rappelons qu'à tout moment Δ contient les sommets pour lesquels `distances` et `passage` contiennent toute l'information relative à un plus court chemin qui les relie à la source. Ceci est vrai au début de l'algorithme, nous en aurons donc terminé si nous arrivons à expliciter une étape qui conserve cette condition et qui augmente l'ensemble Δ.

attention ! c'est ici le cœur de l'algorithme !

Nous choisissons un sommet s de C (ceux qui ne font pas encore partie de Δ) qui minimise `distances`[s], nous le supprimons de C et l'ajoutons à Δ. Puis pour chaque sommet t qui reste dans C, nous mettons à jour `distances`[t] et `passage`[t] de la façon suivante : si `distances`[s] $+ \varphi(s,t) <$ `distances`[t], alors nous remplaçons `distances`[t] par `distances`[s] $+ \varphi(s,t)$ et `passage`[t] par s.

Il n'est pas évident qu'ainsi défini notre algorithme fonctionne. C'est pourtant le cas, mais notre hypothèse de récurrence est insuffisante. Voici la propriété qui nous permettra de conclure, si nous arrivons à prouver qu'elle est conservée durant tout le déroulement de l'algorithme :

> Les tableaux `distances` et `passage` permettent d'obtenir pour tous les points $u \in C$ un plus court chemin qui mène de la source à u en ne passant que par des points de Δ.
>
> Les tableaux `distances` et `passage` permettent d'obtenir pour tous les points $u \in \Delta$ un plus court chemin qui mène de la source à u.

L'initialisation des deux tableaux a été justement faite afin de vérifier cette hypothèse au début de l'algorithme, quand $\Delta = \{\text{source}\}$.

À la fin de l'algorithme, $\Delta = S$, et nous avons donc bien le résultat attendu.

Reste à prouver qu'à chaque étape nous conservons la propriété en question.

✧

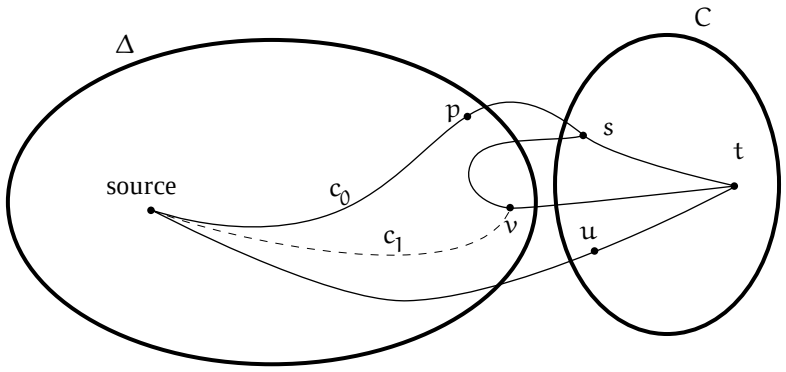


Figure 8.1: Une étape de l'algorithme de Dijkstra

Nous supposons donc que notre condition est vérifiée à un moment donné de l'algorithme. Nous choisissons un sommet s de C qui minimise le tableau `distances`. Nous allons vérifier notre propriété pour s d'une part (on travaille sur la deuxième proposition de l'hypothèse de récurrence), puis pour tout autre sommet $t \in C$ (pour la première proposition de l'hypothèse de récurrence).

Soit $p =$ `passage`[s]. D'après l'hypothèse, il existe un chemin $c_0 = (\text{source}, \ldots, p, s)$ qui ne passe que par des sommets de Δ (sauf s) et qui réalise le minimum de distance. Montrons qu'il s'agit là d'un chemin minimal de la source à s.

Sinon, on pourrait trouver un chemin plus court, qui passe par un sommet $t \in C$ ($t \neq s$). Un tel chemin débutant à la source et terminant à t entre dans C pour la première fois par un certain sommet u (il retournera éventuellement dans Δ, peu importe). Alors, comme toutes les arêtes sont de poids positifs, notre chemin qui va de la source à s via u puis t aura une longueur supérieure ou égale à la longueur du chemin de la source à u, ce dernier chemin ne passant que par des sommets de Δ. Notre choix de s assure que la longueur de c_0 est plus petite : passer par t (donc u) ne peut que rallonger le chemin. C'est gagné !

Occupons nous maintenant d'un sommet t de C, distinct de s.

Il s'agit de mettre à jour `distances`$[t]$ et `passage`$[t]$ de telle sorte que notre condition reste vérifiée. De deux choses l'une : ou bien pour trouver un plus court chemin de la source à t en ne passant que par des points du nouveau Δ, c'est-à-dire l'ancien auquel on a ajouté s, on doit passer par s, ou bien il n'y a rien à modifier, et c'est fini.

Considérons donc un tel chemin qui passerait par s. Je dis qu'il ne devrait pas retourner ensuite dans Δ, sans quoi il en ressortirait par un certain v. Mais v ayant été incorporé dans Δ avant s, le chemin c_0 utilisé pour aller de la source à s est plus long qu'un certain chemin c_1 qui va de la source à v. Ainsi il aurait été plus court de ne pas passer par s (voir la figure).

on ajoute le poids de l'arête (s, t)

Nous sommes donc certain qu'un plus court chemin qui passe par s est de la forme $(\text{source}, \ldots, s, t)$, et l'algorithme de Dijkstra opère alors exactement la mise à jour nécessaire.

✥

On voit pourquoi la dernière étape est inutile : il ne reste plus qu'un sommet s dans C, un plus court chemin de la source à s qui ne passe que par des sommets de $\Delta = S \setminus \{s\}$ est bien sûr un plus court chemin.

8.3 Implémentation de l'algorithme de Dijkstra en Caml

Rappelons tout d'abord notre implémentation de $\mathbb{N} \cup \{+\infty\}$.

Programme 8.1 Les entiers étendus

```
1 (* on introduit un type correspondant à N U { +infini } *)
2 type entier = Infini | N of int;;
3
4 let inférieur x y = match x,y with
5       Infini,Infini -> true
6     | Infini,N(_) -> false
7     | N(_),Infini -> true
8     | N(p),N(q) -> p <= q;;
9
10 let strict_inférieur x y = match x,y with
11       Infini,Infini -> false
12     | Infini,N(_) -> false
13     | N(_),Infini -> true
14     | N(p),N(q) -> p < q;;
15
16 let plus x y = match x,y with
17       Infini,Infini -> Infini
18     | Infini,N(_) -> Infini
19     | N(_),Infini -> Infini
20     | N(p),N(q) -> N(p + q);;
```

On trouvera page suivante le programme Caml qui implémente l'algorithme de Dijkstra.

Programme 8.2 L'algorithme de Dijkstra

```
(*============================================================================*)
(*                                                                            *)
(*          recherche du chemin le plus court d'un sommet fixé aux autres     *)
(*                          algorithme de Dijkstra                            *)
(*                                                                            *)
(*============================================================================*)

load_object "Tas";;
#open "Tas";;

(* dijkstra g Sommet(s) renvoie une paire de deux fonctions distance,chemin     *)
(* distance Sommet(but) donne la plus courte distance de Sommet(s) à Sommet(b)  *)
(* chemin Sommet(but) décrit un chemin qui réalise cette plus courte distance   *)
(* bien sûr, l'algorithme de Dijkstra nous interdit les distances négatives     *)

let dijkstra g (Sommet(nom_source) as source) =
    let n = list_length g
    in
    let distances = make_vect n Infini
    and passage = make_vect n source
    and m,sl = matrice_d_incidence_et_sommets g Infini (function x -> N(x))
    in
    let indice_source = index nom_source sl
    and sv = vect_of_list sl
    (* on fixe le type du tas, et sa taille maximale *)
    and tas = liste_en_tas  ( fun (Sommet a) (Sommet b)
                                -> strict_inférieur distances.(index a sl)
                                                    distances.(index b sl) )
                            [source] n
    in
    extrait_minimum tas; (* on vide le tas *)
    (* initialisation *)
    for i = 0 to n-1 do distances.(i) <- m.(indice_source).(i) done;
    for i = 0 to n-1 do if i <> indice_source
                        then insère_dans_tas tas (Sommet(sv.(i))) done;
    for bcl = 1 to n-2 do
        let (Sommet s) = extrait_minimum tas
        in
        let is = index s sl
        and ensemble_C = tas_en_vecteur tas
        in
        for i = 0 to (vect_length ensemble_C) - 1
        do
            let it = index (match ensemble_C.(i) with Sommet(t) -> t) sl
            in
            let somme = plus distances.(is) m.(is).(it)
            in
            if strict_inférieur somme distances.(it) then
            begin
                distances.(it) <- somme;
                passage.(it) <- Sommet(s);
                percolation tas i
            end
        done
    done;
    let distance = function Sommet(s) -> distances.(index s sl)
    and chemin sommet =
            let rec crée_chemin (Sommet s) l =
                let is = index s sl
                in
                if is = indice_source then source :: l
                else crée_chemin (passage.(is)) ((Sommet s) :: l)
            in crée_chemin sommet []
    in distance,chemin;;
```

En ligne 21 on calcule comme on en a l'habitude la matrice qui décrit φ. Les lignes 25–29 permettent d'initialiser la structure de tas qui va implémenter l'ensemble C.

Pourquoi un tas : parce que les opérations que nous avons à effectuer sur C sont justement l'extraction du sommet qui réalise la plus petite distance, ce pour quoi est conçue une file de priorité.

Notons que nous sommes obligés de fournir une liste du type d'objets que doit contenir le tas. C'est la raison pour laquelle, en ligne 29, on fournit la liste `[source]`. Comme cette initialisation est fictive, nous revidons le tas en ligne 31. La relation d'ordre fournie est celle sur les distances : lignes 26–28.

L'initialisation proprement dite de l'algorithme se fait dans les lignes 33–35.

Les lignes 36–55 forment le cœur de l'algorithme, le résultat de `dijkstra` étant comme on en a maintenant l'habitude le couple `(distance,chemin)` qui est créé dans les lignes 56–63, avec une petite récursion pour `chemin`, dont nous avons déjà parlé.

Il n'y a rien de plus dans la boucle principale que ce que nous avons décrit ci-dessus.

Simplement, remarquons l'appel en ligne 52 à la procédure `percolation`, afin de mettre à sa place dans le tas le sommet pour lequel on vient de modifier l'élément correspondant de `distances`. Signalons que l'élément en question ne peut que remonter dans l'arbre, et que cette percolation n'a donc pas d'influence sur les objets que l'on examinera *ensuite* dans la boucle.

reporte toi au chapitre sur les files de priorité

Une petite remarque pour finir : la gymnastique de la ligne 44 est nécessaire puisque nous sommes convenus de ne pas mélanger les sommets et les noms des sommets du graphe.

8.4 Évaluation

L'initialisation, d'après les résultats qu'on a vus pour la gestion des files de priorité, tourne en $O(n)$.

La boucle principale du programme est exécutée $n-2$ fois. Elle demande l'extraction du minimum qui est en $O(\log(n - \texttt{bcl}))$, donc $O(n)$. On en est donc à $O(n \log n)$.

Restent les percolations. Elles coûtent chacune un O du logarithme de la taille du tas au moment où elle se produit. Combien y en a-t-il? pas davantage que d'arêtes dans le graphe, car on les effectue quand (revoir le schéma ci-dessus) le poids de l'arête (s,t) est suffisament petit : à chaque percolation on peut associer l'arête (s,t) qui lui correspond, et qui ne sera plus jamais la cause d'aucune autre percolation. Il y a donc $O(a)$ percolations, qui coûtent chacune $O(\log n)$.

on a déjà dit que a désigne le nombre d'arêtes du graphe

Comme le graphe est connexe, n est lui-même $O(a)$, et finalement le coût de l'algorithme est bien comme annoncé en $O(a \log n)$.

Chapitre 9

Arbre couvrant minimal d'un graphe non orienté

9.1 Présentation du problème

Imaginons-nous une petit région qui ne dispose que d'une centrale électrique. Il s'agit d'acheminer le courant dans toutes les villes. Bien sûr, tout ce qu'il suffit de faire c'est que chaque ville soit finalement reliée à la centrale, pas forcément directement. Enfin, on souhaite pour des raisons évidentes d'économie que la longueur totale des lignes soit la plus petite possible. Le problème peut être résolu à l'aide de la théorie des graphes : c'est celui de l'arbre couvrant minimal, *minimal spanning tree*, en anglais.

cours du jour du cuivre : $ 3100 la tonne

Dans ce chapitre, nous considérons un graphe non orienté et connexe.

On dit qu'un graphe G' est un *sous-graphe* du graphe G si $S' \subset S$, $A' \subset A$ et $\varphi' = \varphi|_{A'}$: tout sommet de G' est un sommet de G et toute arête de G' est une arête de G, de même poids.

Le sous-graphe G' est dit *couvrant* si il est encore connexe et si $S' = S$: ses arêtes relient tous les sommets du graphe. Bien sûr G est lui-même couvrant donc l'ensemble des sous-graphes couvrants est non vide (et fini car S et A le sont).

Le *poids total* d'un graphe non orienté est la somme des étiquettes de toutes ses arêtes. Le problème qu'on se pose est donc précisément de rechercher un sous-graphe couvrant de poids total minimal.

un cycle est un chemin $c = (a_0, \ldots, a_p)$ où $a_0 = a_p$

Démontrons tout d'abord une propriété qui justifiera le titre de ce chapitre :

Théorème 3 *Si toutes les arêtes du graphe sont de poids positif ou nul, il admet un sous-graphe couvrant minimal qui est aussi un arbre, c'est-à-dire un graphe sans cycle.*

✧ Soit en effet G' un sous-graphe couvrant minimal. On a dit que l'ensemble de ses sous-graphes couvrant est non vide et fini, donc on est assuré de l'existence d'un tel sous-graphe G' couvrant minimal.
Si par malheur il contenait un cycle $c = (a_0, \ldots, a_p)$ où $a_0 = a_p$, je dis que chacune des arêtes $(a_0, a_1), \ldots, (a_{p-1}, a_p)$ est de longueur nulle et que le sous-graphe obtenu en supprimant une quelconque de ces arêtes est encore couvrant minimal. Ainsi, en itérant cette simplification tant que possible, on finit par tomber sur un sous-graphe couvrant minimal sans cycle, c'est-à-dire l'arbre couvrant minimal désiré.
En effet si je considère le graphe G'' obtenu en supprimant l'arête $\alpha = (a_k, a_{k+1})$, où $0 \leq k < p$, le poids total de G'' est égal à celui de G' moins le poids de l'arête α. Mais G'' est encore couvrant : l'ensemble des sommets n'a pas bougé, et c reliait entre eux les sommets $a_0, \ldots, a_p$, ce que font encore les chemins restants : $(a_0, \ldots, a_k)$ et $(a_{k+1}, \ldots, a_p)$ car on peut les mettre bout à bout (le graphe est non orienté) et écrire que le chemin $(a_{k+1}, \ldots, a_p = a_0, \ldots, a_k)$ est un chemin dans G''.

Finalement G'' est encore couvrant, alors que G' était couvrant minimal, donc G'' et G ont le même poids et $\varphi(\alpha) = 0$, comme on l'avait annoncé.
✥

Dans toute la suite nous supposons que G est un graphe non orienté, connexe, d'arêtes de longueurs toutes positives ou nulles.

Montrons maintenant un résultat sur l'arbre couvrant minimal, qui sera au cœur des algorithmes que nous allons étudier.

Théorème 4 *Soit* G *un graphe non orienté connexe d'arêtes de longueurs toutes positives ou nulles. Soit* $S = S_1 \cup S_2$ *une partition de ses sommets. Soit* ${}_1A_2$ *l'ensemble des arêtes* (u_1, u_2) *du graphe telles que* $u_1 \in S_1$ *et* $u_2 \in S_2$*. Soit enfin* (u, v) *une arête de* ${}_1A_2$ *de poids minimal. Alors il existe un arbre couvrant minimal de* G *qui contient cette arête* (u, v).

✧ Remarquons tout d'abord qu'on est assuré que ${}_1A_2 \neq \emptyset$ car G est un graphe connexe. Soit donc (u, v) de poids minimal dans ${}_1A_2$, et soit G' un arbre couvrant minimal de G qui ne contienne pas (u, v). Nous allons montrer comment construire à partir de G' un arbre couvrant minimal qui contienne notre arête (u, v) préférée.
Notons tout d'abord que G' contient une et une seule arête $(u', v') \in {}_1A_2$: une, car il est couvrant, une seule, car c'est un arbre. Si en effet il y avait une autre arête $(u'', v'') \in {}_1A_2$, u' et u'' sont connectés par G', ainsi que v' et v'', et on obtiendrait un cycle, ce qui est exclu.

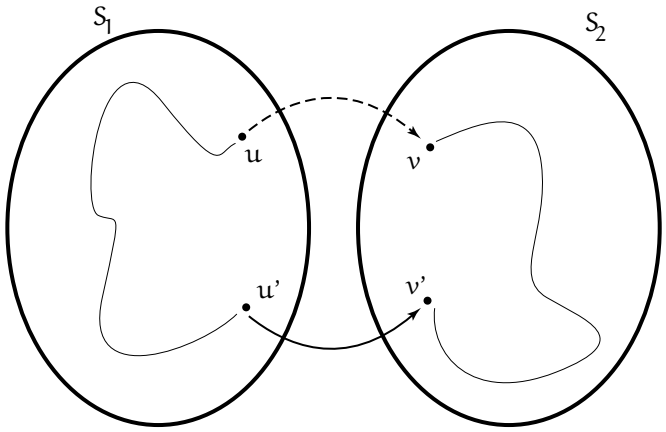


Figure 9.1: Arbre couvrant minimal après partition des sommets

Comme G' est connexe et que (u', v') est la seule arête qui fasse le *pont* entre S_1 et S_2, on est assuré qu'il existe dans G' un chemin qui relie u et u' et un chemin qui relie v et v'. Construisons alors G'' à partir de G' en supprimant l'arête (u', v') et en ajoutant l'arête (u, v).
Comme (u, v) est de poids inférieur à celui de (u', v') (c'est la définition de (u, v)), le poids total de G'' est inférieur au poids total de G'. Il ne reste plus qu'à montrer que G'' est encore couvrant pour que G'' soit l'arbre cherché.
Or c'est bien le cas car les arêtes de G', donc de G'', permettent de relier tout sommet de S_1 à u sans quitter S_1, et tout sommet de S_2 à v sans quitter S_2. En outre il n'y a pas de cycle dans les chemins inclus dans S_1 ni dans les chemins inclus dans S_2, et enfin (u, v) est la seule arête qui fasse le pont entre S_1 et S_2 : ça marche.
✥

9.2 Algorithme de Prim

9.2.1 Description informelle

Prim a publié son algorithme en 1957

L'algorithme de Prim est une application directe du théorème 4 ; il fait partie de la grande famille des algorithmes gloutons.

On initialise un ensemble A' d'arêtes en posant $A' = \emptyset$, et un ensemble S' de sommets en posant $S' = \{u_0\}$, où u_0 est un sommet quelconque du graphe G.

Ensuite, et tant que $S' \neq S$, on itère la manipulation suivante : choisir une arête (u, v) telle que $u \in S'$, $v \notin S'$ et qui soit de poids minimal, l'ajouter à A', et ajouter alors v à S'.

Quand on arrive à la fin du processus, $S' = S$, et A' est l'ensemble des arêtes d'un arbre couvrant minimal.

L'algorithme termine car à chaque étape S' augmente d'un, et, pour le reste, il s'agit d'une application directe du théorème 4, rien de plus.

9.2.2 Implémentation en Caml

Pour écrire l'algorithme en Caml, nous utiliserons à nouveau notre définition des entiers étendus $\mathbb{N} \cup \{+\infty\}$, que nous rappelons ci-dessous.

Programme 9.1 Les entiers étendus

```
1 (* on introduit un type correspondant à N U { +infini } *)
2 type entier = Infini | N of int;;
3
4 let inférieur x y = match x,y with
5       Infini,Infini -> true
6     | Infini,N(_) -> false
7     | N(_),Infini -> true
8     | N(p),N(q) -> p <= q;;
9
10 let strict_inférieur x y = match x,y with
11       Infini,Infini -> false
12     | Infini,N(_) -> false
13     | N(_),Infini -> true
14     | N(p),N(q) -> p < q;;
15
16 let plus x y = match x,y with
17       Infini,Infini -> Infini
18     | Infini,N(_) -> Infini
19     | N(_),Infini -> Infini
20     | N(p),N(q) -> N(p + q);;
```

On commencera par calculer, comme on l'a expliqué plus haut, la matrice `m` d'incidence du graphe. Pour suivre l'algorithme de Prim, nous allons tout au long de son déroulement conserver dans différents tableaux l'information nécessaire.

`déjà_choisi` est un tableau de booléens qui dira si oui ou non un sommet est déjà dans l'ensemble S'. Au départ seul `déja_choisi`[0] est vrai, car nous choisissons ainsi u_0 ;

`plus_proche` est un tableau qui contient pour chaque sommet v qui n'est pas encore choisi le numéro du sommet u de S' tel que (u, v) soit de poids minimal. Au départ il contient donc uniquement des 0, qui est le numéro du sommet u_0 ;

`à_distance_de` enfin conserve la longueur de l'arête (u, v) précédente. Au départ il contient donc simplement les `m.(0).(i)`.

Observons d'autre part que l'algorithme de Prim demande l'exécution de $(n-1)$ étapes : à chaque étape on ajoute un nouveau sommet à S' qui au départ n'en contenait qu'un et qui à l'arrivée en a n ; ou encore : à chaque étape on ajoute une nouvelle arête à A' qui au départ est vide et qui à l'arrivée en comprend $(n-1)$, puisqu'on a un arbre connexe à n sommets.

démontre ça ! – par récurrence, peut-être ?

Pour créer la boucle nous allons écrire une procédure récursive `boucle_principale` à deux arguments, la liste des arêtes de A', et sa taille, qui se terminera quand cette taille vaudra justement $(n-1)$.

À chaque étape, la recherche de l'arête à ajouter à A' est une très classique recherche de minimum dans un tableau ; plus difficile est la mise à jour des tableaux `plus_proche` et `à_distance_de`. Quand on ajoute l'arête (u,v) à A' et donc le sommet v à S', pour un sommet w quelconque qui reste dans S', on n'aura à modifier les éléments correspondants dans ces tableaux que si $\phi(v,w)$ améliore la "distance" de w à S', c'est-à-dire si et seulement si $\phi(v,w) <$ `à_distance_de`$[i_w]$.

On verra page suivante le listing correspondant en Caml.

Quelques commentaires pour conclure :

lignes 2–12 : l'initialisation des différents tableaux ;

lignes 14–32 : la procédure `crée_graphe_depuis_liste` prend une liste d'arêtes et renvoie le graphe correspondant. C'est sans doute la plus compliquée de tout ce programme, mais elle n'a rien à voir avec notre sujet : l'algorithme de Prim. On pourra donc la sauter en première lecture ;

lignes 35–40 : la recherche du minimum. On fournit en arguments le nombre d'éléments déjà balayés, le minimum courant et son indice, et donc au départ 1 (car le premier sommet est toujours dans S'), `Infini` et 0 ;

lignes 42–53 : la mise à jour des tableaux `plus_proche` et `à_distance_de`, qui ne concerne que les points pas encore choisis. Prend deux arguments : le nombre d'éléments déjà mis à jour et l'indice du sommet nouvellement ajouté à S' ;

lignes 55–64 : la boucle principale prend deux arguments, la liste courante des arêtes de A' et le nombre d'éléments dans cette liste.

9.2.3 Évaluation

On a déjà dit que la boucle principale est en $(n-1)$ étapes.

Chaque étape demande la recherche d'un minimum, en $O(n)$, et la mise à jour des différents tableaux, également en $O(n)$.

Finalement, on peut en conclure que l'algorithme de Prim tourne en $O(n^2)$.

Programme 9.2 L'algorithme de Prim

```
let prim g =
    let n = list_length g
    in                      (* phase d'initialisation *)
    let plus_proche = make_vect n 0
    and déjà_choisi = make_vect n false
    and à_distance_de = make_vect n (N 0)
    and m,sl = matrice_d_incidence_et_sommets g Infini (function x -> N(x))
    in
    let sv = vect_of_list sl
    in
    for i = 1 to n-1 do à_distance_de.(i) <- m.(0).(i) done;
    déjà_choisi.(0) <- true;
    (* pour construire le graphe résultat *)
    let crée_graphe_depuis_liste l =
            let rec symétrique = function (u,v) -> (v,u)
            and gonfle_graphe graphe = function
                  [] -> graphe
                | arc :: q
                    -> gonfle_graphe
                         (ajoute_arete arc
                             (ajoute_arete (symétrique arc) graphe)) q
            and ajoute_arete (i1,i2) g =
                let s1 = Sommet(sv.(i1)) and s2 = Sommet(sv.(i2))
                and v = match m.(i1).(i2) with N(nv) -> nv | Infini -> 999999
                in
                match g with
                      [] -> [ (s1,[(s2,v)]) ]
                    | (((Sommet nom),l) as a) :: q
                        -> if i1 = (index nom sl) then (s1,(s2,v)::l)::q
                           else a::(ajoute_arete (i1,i2) q)
            in
            gonfle_graphe [] l
    in
    (* on revient à l'algorithme de Prim proprement dit *)
    let rec arête_la_moins_chère i min imin =
        if i = n then imin
        else    if (not déjà_choisi.(i))
                   & (strict_inférieur à_distance_de.(i) min)
                then arête_la_moins_chère (i+1) (à_distance_de.(i)) i
                else arête_la_moins_chère (i+1) min imin
    and
    mise_à_jour i k =
        if i < n then
        begin
            if (not déjà_choisi.(i))
               & (strict_inférieur m.(k).(i) à_distance_de.(i))
            then
            begin
                à_distance_de.(i) <- m.(k).(i);
                plus_proche.(i) <- k
            end;
            mise_à_jour (i+1) k
        end
    in
    let rec boucle_principale liste k =
        if k < n - 1 then
        begin
            let imin = arête_la_moins_chère 1 Infini 0
            in
            déjà_choisi.(imin) <- true;
            mise_à_jour 1 imin;
            boucle_principale ((imin,plus_proche.(imin)) :: liste) (k+1)
        end
        else liste
    in
    crée_graphe_depuis_liste (boucle_principale [] 0);;
```

9.3 Algorithme de Kruskal

9.3.1 Description de l'algorithme

Kruskal publie son algorithme en 1956. Il s'agit encore d'un algorithme glouton, qui se prouve aisément par le théorème 4. Cependant il part d'un point de vue radicalement différent de celui de Prim.

broups

Au lieu en effet de privilégier un sommet et de faire gonfler l'arbre couvrant en partant de ce sommet fixé, Kruskal propose de partitionner en singletons l'ensemble des sommets, et, en utilisant les arêtes de l'arbre en ordre croissant de longueur, de réunir petit à petit des classes de sommets jusqu'à n'avoir plus qu'une seule classe : l'arbre couvrant minimal est alors atteint.

j'espère que cette description, elle, n'est pas trop gonflante. . .

Plus précisément, on trie les arêtes de A, en formant une file de priorité (un tas) $\mathcal{T}$. On crée une partition en singletons $\mathcal{P}$ de l'ensemble S des sommets. On initialise enfin une liste d'arêtes $\mathcal{L}$ à vide.

Ensuite, tant que $\mathcal{P}$ n'est pas réduit à une seule classe, on effectue les opérations suivantes : on extrait l'arête (u, v) la moins longue du tas $\mathcal{T}$; si u et v sont dans la même classe de $\mathcal{P}$, tant pis, on boucle ; sinon on réunit les deux classes de u et v en une seule, on ajoute l'arête (u, v) à la liste finale $\mathcal{L}$ d'arêtes, et on boucle.

À la fin de l'algorithme $\mathcal{L}$ contient une liste d'arêtes qui forment un arbre couvrant minimal.

9.3.2 Implémentation en Caml

On trouvera le listing correspondant page ci-contre.

Quelques commentaires :

lignes 1–5 : chargement des bibliothèques qui nous seront utiles ;

lignes 11–18 : `crée_liste_des_arêtes` crée la liste des arêtes en tant que $(u, v, \varphi(u, v))$ dont on fera le tas $\mathcal{T}$ en ligne 45 avec la relation d'ordre adéquate ;

lignes 20–36 : on retrouve ici la même procédure que dans l'algorithme de Prim, pour reconstituer le graphe final à partir de la liste d'arêtes correspondante ;

lignes 38–40 : on crée ici la liste des sommets, par une petite procédure récursive, afin de se préparer à la création, ligne 46, de la partition $\mathcal{P}$ initiale ;

lignes 49–62 : la boucle principale, qui prend en arguments la liste courante $\mathcal{L}$ et sa taille. On quitte la boucle quand cette taille est égale à $(n - 1)$, puisque c'est le nombre d'arêtes d'un arbre couvrant. La ligne 56 teste si les deux extrémités de l'arête extraite du tas $\mathcal{T}$ en ligne 52 sont ou non dans la même classe de $\mathcal{P}$. Si oui, on fusionne, et on boucle avec une liste $\mathcal{L}$ agrandie (ligne 58) ; sinon on boucle avec la liste initiale, ligne 60.

9.3.3 Évaluation

La création du tas se fait en $O(a)$, où a est le nombre d'arêtes. Celle de la partition en $O(n)$.

il est temps de rendre des comptes

On a déjà dit qu'il y avait $(n - 1)$ étapes dans la boucle principale. Chaque étape demande une recherche-fusion dans la partition, et une extraction du tas. L'extraction se réalise en $O(\log a)$, à toutes bonnes fins de percolation.

Pour ce qui est de la partition, on a déjà évalué l'ensemble de toutes les opérations effectuées : on part en effet d'une partition en n singletons et on arrive à une partition en une seule classe. Nous savons que tout cela coûte $O(a)$, pour des valeurs raisonnables de n (revoir la discussion correspondante dans le chapitre sur les partitions).

Or $n - 1 \leq q \leq n^2$, donc en particulier $O(\log a)$ est $O(\log n)$. Au total, l'algorithme de Kruskal tourne donc en $O(a + n \log n)$.

Programme 9.3 L'algorithme de Kruskal

```
load_object "Partitions";;
#open "Partitions";;

load_object "Tas";;
#open "Tas";;

(* kruskal g renvoie un nouveau graphe (en fait un arbre) qui constitue *)
(* un arbre de recouvrement minimal du graphe non orienté donné         *)

let kruskal g =
    let rec crée_liste_des_arêtes accu_final g =
            let rec dévide accu source = function
                      [] -> accu
                    | (s,v) :: q -> dévide ((source,s,v)::accu) source q
            in
            match g with
              [] -> accu_final
            | (s,l) :: q -> dévide (crée_liste_des_arêtes accu_final q) s l
    in
    let crée_graphe_depuis_liste l =
            let rec symétrique = function (u,v,w) -> (v,u,w)
            and
            gonfle_graphe graphe = function
                  [] -> graphe
                | arc :: q -> gonfle_graphe
                                (ajoute_arete arc
                                    (ajoute_arete (symétrique arc) graphe)) q
            and
            ajoute_arete =
                function ((((Sommet nom1) as s1),s2,v) as arc) -> function
                  [] -> [ (s1,[(s2,v)]) ]
                | (((Sommet n) as a),l) :: q ->
                    if eq_string n nom1 then (a,(s2,v)::l)::q
                    else (a,l)::(ajoute_arete arc q)
            in
            gonfle_graphe [] l
    in
    let rec crée_liste_des_sommets accu = function
          [] -> accu
        | (s,l) :: q -> crée_liste_des_sommets (s::accu) q
    in
    let la = crée_liste_des_arêtes [] g
    and ls = crée_liste_des_sommets [] g
    in
    let tas = liste_en_tas  ( fun (_,_,p) (_,_,q) -> p < q ) la (list_length la)
    and les_nœuds = liste_en_partition ls (function s -> (index s ls) )
    and ns = list_length ls
    in
    let rec boucle_principale liste n =
        if n < ns - 1 then
        begin
            let (s1,s2,v) = extrait_minimum tas
            in
            let n1,n2 = (trouve_paquet les_nœuds s1),(trouve_paquet les_nœuds s2)
            in
            if n1 <> n2 then
                (  fusionne_paquets les_nœuds n1 n2;
                   boucle_principale ((s1,s2,v) :: liste) (n+1)
                )
            else boucle_principale liste n
        end
        else liste
    in
    crée_graphe_depuis_liste (boucle_principale [] 0);;
```

Partie III

Quelques algorithmes de géométrie combinatoire

Chapitre 10

Enveloppe convexe d'un ensemble de points du plan

10.1 Rappels sur la convexité

les Anglo-Saxons disent convex hull

Rappelons qu'une partie X du plan est dite convexe si elle est vide ou si pour tout couple (a, b) de points de X le segment $[ab]$ est tout entier inclus dans X. On démontre alors très simplement que toute intersection de convexes est convexe, et on est conduit tout naturellement à définir l'enveloppe convexe d'une partie quelconque non vide X du plan comme l'intersection de tous les convexes qui contiennent X, puisque bien sûr le plan lui-même est convexe. Nous noterons dans la suite $\mathrm{Conv}(X)$ l'enveloppe convexe de X, et en cas de besoin $\partial\mathrm{Conv}(X)$ sa frontière.

Nous nous intéresserons ici plus particulièrement du cas où X est un ensemble fini de n points du plan. On notera qu'alors $\partial\mathrm{Conv}(X)$ est un polygone (convexe, évidemment), et rechercher l'enveloppe convexe de X reviendra à donner la liste ordonnée (en général dans le sens trigonométrique) des sommets successifs de ce polygone frontière.

On va voir dans la suite — par deux approches totalement différentes — des algorithmes qui permettent de reconstituer l'enveloppe convexe d'un ensemble de n points du plan en $O(n \log n)$.

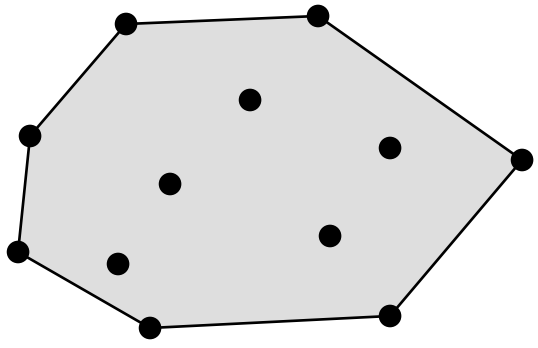
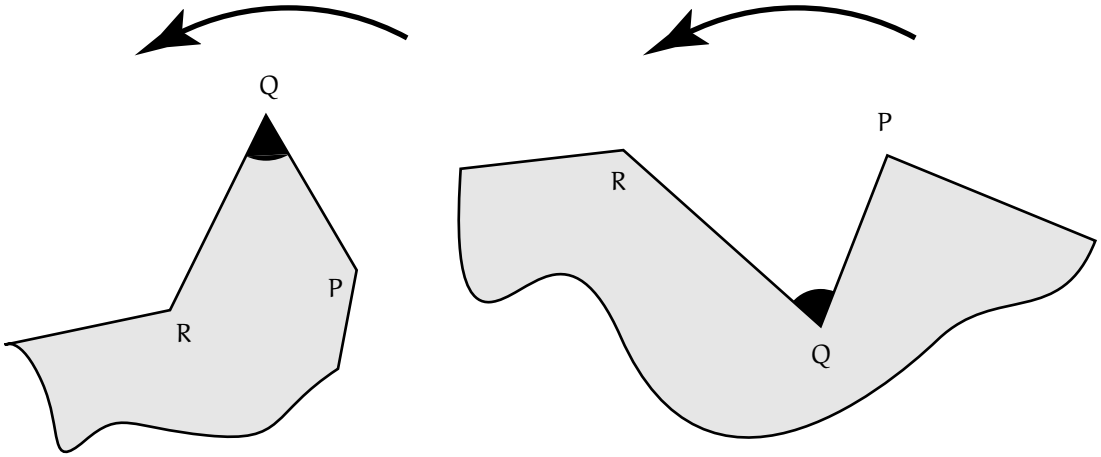


Figure 10.1: Un exemple d'enveloppe convexe dans le plan

10.2 Algorithme de Graham-Andrew

10.2.1 La marche de Graham

La première approche repose sur la classification des angles que forment deux côtés successifs d'un polygone : on distinguera en effet les angles *saillants* et les angles *rentrants* (voir la figure suivante).



Angle saillant Angle rentrant

Figure 10.2: Classification des angles au sommet d'un polygone

La définition de cette classification suppose qu'on utilise une orientation appropriée. Considérons par exemple un polygone convexe, et parcourons la suite de ces sommets dans l'ordre croissant de leurs angles polaires (dans le sens trigonométrique, donc) : tout point intérieur au polygone reste alors toujours *à gauche* de notre parcours, ou, ce qui revient au même, tout angle au sommet du polygone est un angle saillant. Sur la figure précédente, à droite, le fait que PQR soit un angle rentrant suffit à prouver que Q ne fait pas partie de l'enveloppe convexe de l'ensemble des points $\{\ldots, P, Q, R, \ldots\}$. C'est la remarque qui est au cœur de l'algorithme de Graham.

En outre, il est très facile de tester si un angle est saillant ou rentrant : PQR est saillant si et seulement si $\det(\overrightarrow{PQ}, \overrightarrow{QR})$ est positif, et on s'en sort avec une comparaison, deux multiplications et une soustraction.

une bagatelle !

Précisons maintenant l'algorithme de Graham proprement dit, c'est-à-dire ce qu'on appelle traditionnellement la marche de Graham. On commence par ranger les n points par ordre croissant de leurs angles polaires (on fixe une fois pour toutes l'origine en un point intérieur à $\mathrm{Conv}(X)$; il suffit pour cela de prendre le milieu de deux points de X, par exemple), ou de leurs distances à l'origine, s'ils ont même angle polaire. On va les numéroter suivant l'ordre ainsi défini, mais en commençant par un point p_0 dont on peut être sûr qu'il fait partie de $\partial\mathrm{Conv}(X)$. Il suffit pour cela de prendre un point extrémal, par exemple un de ceux qui ont la plus grande abscisse. On a finalement numéroté les points $p_0, p_1, \ldots, p_{n-1}$. On convient aussi d'utiliser une numérotation *modulo* n, ce qui revient par exemple à écrire $p_n = p_0$, ou $p_{-1} = p_{n-1}$.

tu parles d'une tradition, ça date de 1972 !

L'algorithme de Graham est alors le suivant : on va avancer dans la liste des points, jusqu'à retrouver p_0, en supprimant au fur et à mesure les points qui ne font pas partie de $\partial\mathrm{Conv}(X)$.

À la fin du processus, la liste des points restants contient la suite ordonnée des sommets du polygone-frontière de l'enveloppe convexe cherchée.

on numérote toujours modulo n

À chaque étape on considère trois points consécutifs p_k, p_{k+1}, p_{k+2}. De deux choses l'une :

$p_kp_{k+1}p_{k+2}$ est un angle saillant et on avance : $k \leftarrow k+1$;

$p_kp_{k+1}p_{k+2}$ est un angle rentrant et alors le sommet p_{k+1} ne peut être sur le polygone cherché. On le supprime donc de la liste, mais on est obligé de revenir en arrière, et on pose donc $k \leftarrow k-1$.

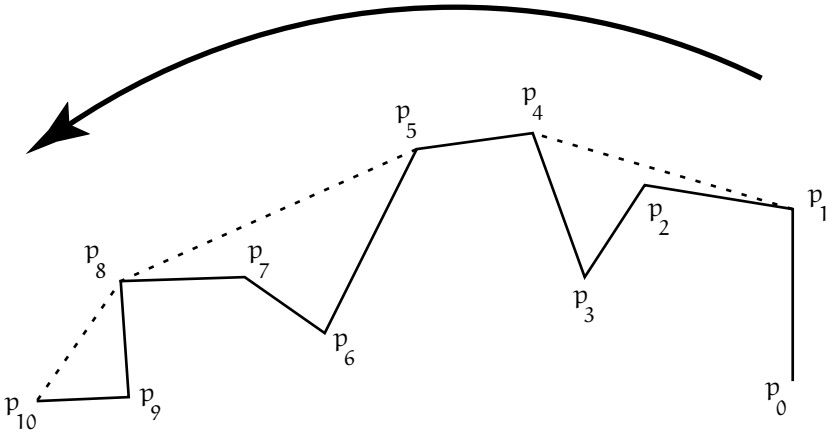


Figure 10.3: La marche de Graham

Dans l'exemple de la figure ci-dessus, voici la liste des triplets successivement considérés : $p_0p_1p_2$, $p_1p_2p_3$, $p_2p_3p_4$ (on supprime p_3), $p_1p_2p_4$ (on supprime p_2), $p_0p_1p_4$, $p_1p_4p_5$, $p_4p_5p_6$, $p_5p_6p_7$ (on supprime p_6), $p_4p_5p_7$, $p_5p_7p_8$ (on supprime p_7), $p_4p_5p_8$, $p_5p_8p_9$, $p_8p_9p_{10}$ (on supprime p_9), etc.

On est sûr que l'algorithme termine car p_0 doit rester dans la liste finale, et car à chaque étape, soit on est dans le premier cas, et on avance clairement, soit on est dans le second cas, mais c'est qu'on supprime un point, et ça ne peut arriver plus de n fois, évidemment ! Donc

id est *en* $O(n)$

non seulement l'algorithme termine, mais en plus il est linéaire.

Bizarre, non? on avait annoncé un algorithme en $O(n \log n)$... Simplement, il faut penser au coût du tri de la phase de préparation de l'algorithme, pour retrouver le $O(n \log n)$ prévu.

10.2.2 L'algorithme de Graham est optimal

on devrait savoir...

On sait qu'un tri en $O(n \log n)$ est optimal. Pour prouver que l'algorithme de Graham est optimal, il suffit donc de montrer que si on sait résoudre le problème de la recherche de l'enveloppe convexe en un temps $T(n)$, on peut en déduire un algorithme de tri qui tourne dans le même temps : cela prouvera que $T(n)$ est au moins $O(n \log n)$.

Supposons donc résolu le problème de l'enveloppe convexe, et déduisons en un algorithme de tri.

Soit n réels x_0, x_1, ..., x_{n-1}, auxquels on fait correspondre n points m_i sur l'axe des x. Soit alors, pour chaque i, p_i le point d'abscisse x_i (comme m_i) de la parabole d'équation $y = x^2 + 1$. Rechercher l'enveloppe convexe des points p_i c'est alors fournir *dans l'ordre trigonométrique* les sommets consécutifs du polygone-frontière, et c'est donc numéroter les p_i en ordre croissant de leurs abscisses. Bref, c'est trier nos n réels. CQFD.

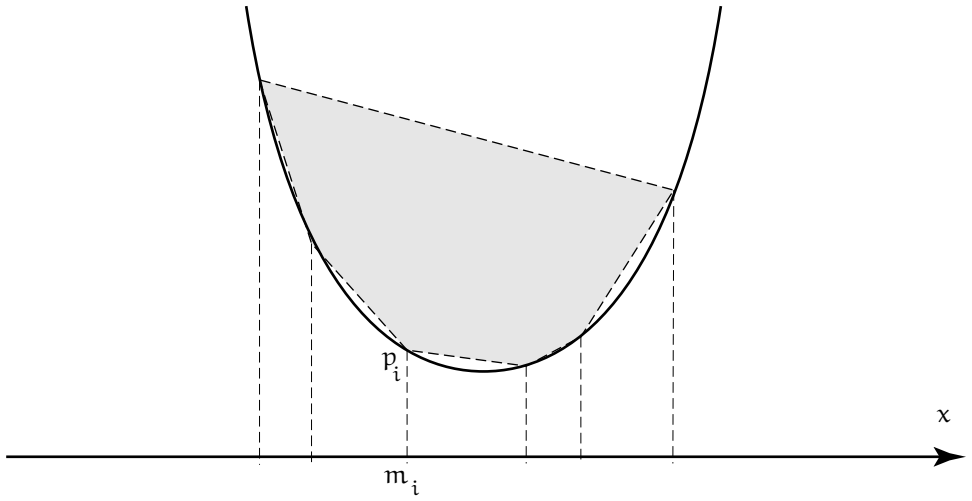


Figure 10.4: Optimalité de l'algorithme de Graham

Cette méthode de transformation d'un problème en un autre est une méthode privilégiée pour majorer ou minorer le coût optimal d'un algorithme, et peut être utilisée pour prouver l'optimalité de nombreux algorithmes classiques. On prendra toutefois garde à ne pas oublier de vérifier que la transformation elle-même ne requiert pas un temps d'un ordre égal ou, pire, supérieur aux temps qu'on est en train de comparer. Dans le cas présent, la transformation se fait en un temps linéaire, et notre démonstration est bien correcte.

10.2.3 L'amélioration de l'algorithme de Graham par Andrew

On peut toutefois reprocher à l'algorithme de Graham les calculs qu'imposent le tri initial des points selon leurs angles polaires. Sensible à ce problème, Andrew propose en 1979 une petite modification de l'algorithme de Graham.

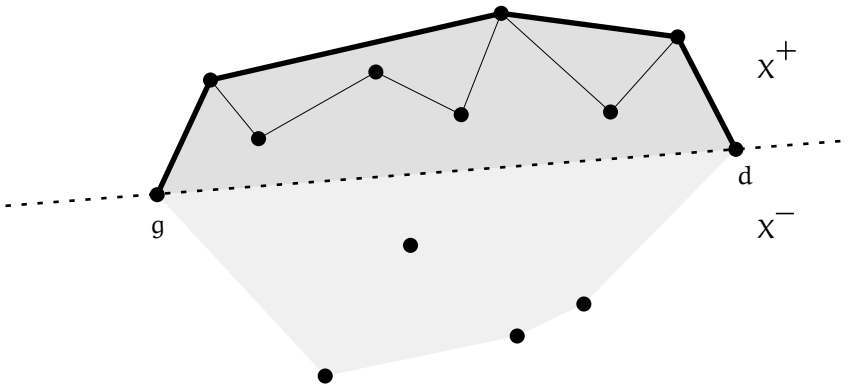


Figure 10.5: L'idée de Andrew

Il considère les points g et d extrémaux en abscisses : g est le point le plus à gauche, et d le plus à droite. Bien sûr, il est certain que g et d font partie de $\partial \mathrm{Conv}(X)$. Andrew trace alors la droite gd et partage ainsi X en deux parties X^+ et X^- auxquelles il va appliquer séparément une

les points qui seraient sur (gd) *peuvent être ignorés*

marche de Graham un peu modifiée. Le segment $[gd]$ est un des côtés du polygone $\partial\text{Conv}(X^+)$, les autres côtés forment ce que Andrew baptise la frontière supérieure de l'enveloppe convexe de X, qui est limitée par g et d. Il définit de même la frontière inférieure.

Andrew annonce alors qu'il suffit d'appliquer à chacune des parties X^+ et X^- la marche de Graham habituelle, sauf qu'on trie les points tout simplement par abscisses !

En réalité, Andrew utilise *exactement* la même méthode que Graham, à un passage près par la géométrie projective, où il envoie à l'infini le pôle qu'utilisait Graham. C'est le même algorithme, on s'est simplement arrangé — par un changement de *repère* (projectif) — pour avoir des calculs plus simples.

10.3 Stratégie diviser pour régner

Nous allons maintenant décrire un algorithme de résolution du même problème de l'enveloppe convexe d'un ensemble X de n points, mais qui utilise cette fois la stratégie *diviser pour régner*. On verra qu'il est comparable en efficacité au précédent, puisqu'également en $O(n \log n)$.

Bien entendu, fidèle à la stratégie, nous découpons l'ensemble initial en deux parties de tailles à peu près égales, X_1 et X_2. On applique alors l'algorithme récursivement sur chacune de ces moitiés, obtenant ainsi les deux polygones convexes $\partial\text{Conv}(X_1)$ et $\partial\text{Conv}(X_1)$. Tout le problème réside donc dans la recherche de l'enveloppe convexe de la réunion de deux polygones convexes, et c'est ce à quoi nous allons nous intéresser maintenant.

10.3.1 Algorithme de Shamos

Shamos imagine la méthode suivante en 1978.

Soit donc P_1 et P_2 deux polygones convexes, ayant respectivement n_1 et n_2 sommets, on souhaite construire l'enveloppe convexe de leur réunion.

Shamos propose de considérer un point p intérieur à P_1. Pour cela, il suffit de considérer le centre de gravité de trois des sommets de P_1.

Alors, de deux choses l'une :

p est intérieur à P_2 : voir la première partie de la figure 10.6, page suivante. On dispose des listes ordonnées des sommets des deux polygones P_1 et P_2. Observons que comme p est intérieur aux deux polygones, ce sont aussi des listes de sommets ordonnées par leurs angles polaires (pour le pôle p). On peut les fusionner en un temps $O(n_1 + n_2)$ pour former la liste des sommets de $P_1 \cup P_2$ ordonnée par ordre croissant des angles polaires. Il suffit alors d'utiliser l'algorithme de Graham pour récupérer $\text{Conv}(P_1 \cup P_2)$.

p est extérieur à P_2 : voir la seconde partie de la figure 10.6, page suivante. Cette fois, on peut (en un temps de l'ordre de $O(n_2)$) déterminer les deux sommets u et v pour lesquels l'angle polaire ps est extrémal pour $s \in P_2$. On a ainsi coupé le contour de P_2 en deux parties, et l'on abandonne celle qui correspond aux sommets qui ne feront pas partie de l'enveloppe convexe cherchée. On procède alors comme précédemment, en fusionnant la liste ordonnée des sommets de P_1 et celle de ceux des sommets de P_2 qu'on a conservés, puis en appliquant l'algorithme de Graham.

Le lecteur attentif aura pu se demander comment déterminer rapidement si un point p donné est intérieur ou non au polygone convexe P_2. La réponse est simple : on considère la droite horizontale passant par p. Si elle contient un des côtés du polygone, p est extérieur au polygone et on a terminé. Sinon, on cherche ses points d'intersection avec les différents côtés du polygone. On en trouvera deux. Si p est entre eux c'est que p est intérieur au polygone, et extérieur sinon. Tout cela se fait en $O(n_2)$.

Au total, puisque $n_1 + n_2 \leq n$, l'étape de fusion tourne en $O(n)$ et notre algorithme en $O(n \log n)$.

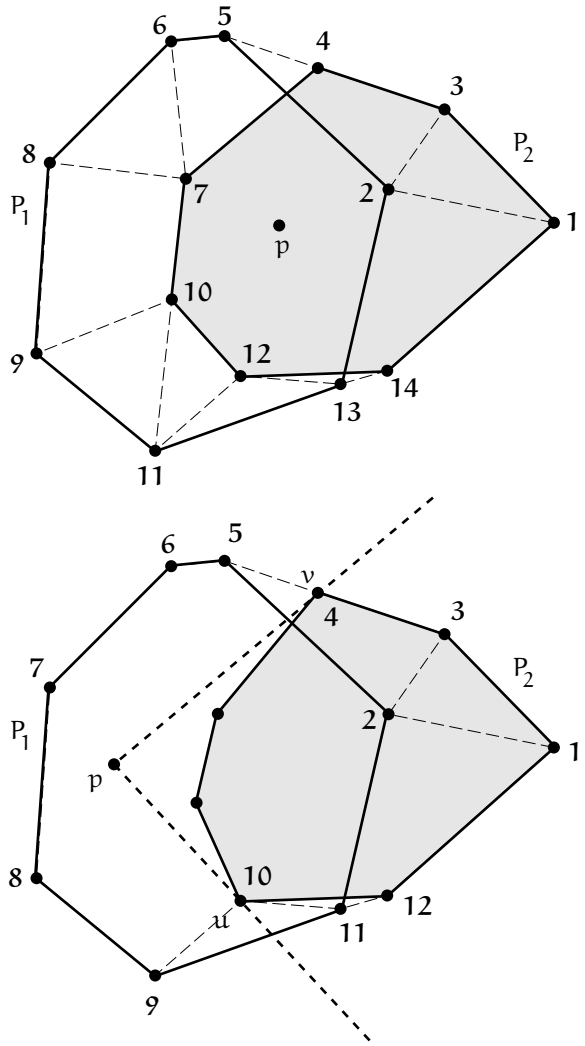


Figure 10.6: Les deux cas : p est intérieur ou extérieur à P_2

10.3.2 Algorithme de Preparata et Hong

Cet algorithme, un peu plus simple, date de 1977.

Il s'agit toujours de déterminer $\mathrm{Conv}(P_1 \cup P_2)$, mais, cette fois, on suppose qu'on a partagé l'ensemble initial en deux parties de même taille par une droite verticale. De cette façon, on est sûr que les intérieurs de P_1 et de P_2 sont disjoints.

Notre intention est de déterminer les deux arêtes de $\partial\mathrm{Conv}(P_1 \cup P_2)$ qui ne font pas déjà partie des polygones eux-mêmes, à savoir ce qu'on appelle les ponts inférieur et supérieur. Se référant à la figure 10.7, nous allons expliquer comment trouver le pont inférieur, noté ici [5f].

On part du point le plus à droite du polygone de gauche, à savoir 1, et du point le plus à gauche du polygone de droite, a. Depuis 1, on trace les arêtes [1a], [1b], [1c], mais pas [1d] qui nous ferait remonter.

on tourne dans le sens rétrograde quand on s'appuie à gauche

On s'appuie maintenant sur c, et on trace [1c], [2c], [3c], [4c], mais pas [5c] qui nous ferait rebrousser chemin.

on tourne dans le sens direct quand on s'appuie à droite

Et on itère, s'appuyant sur 4, traçant [4c], [4d], [4e] et [4f]. On s'appuie enfin sur f, et trace [4f], [5f], où l'on s'arrête car, que l'on s'appuie à gauche ou à droite, on ne peut plus

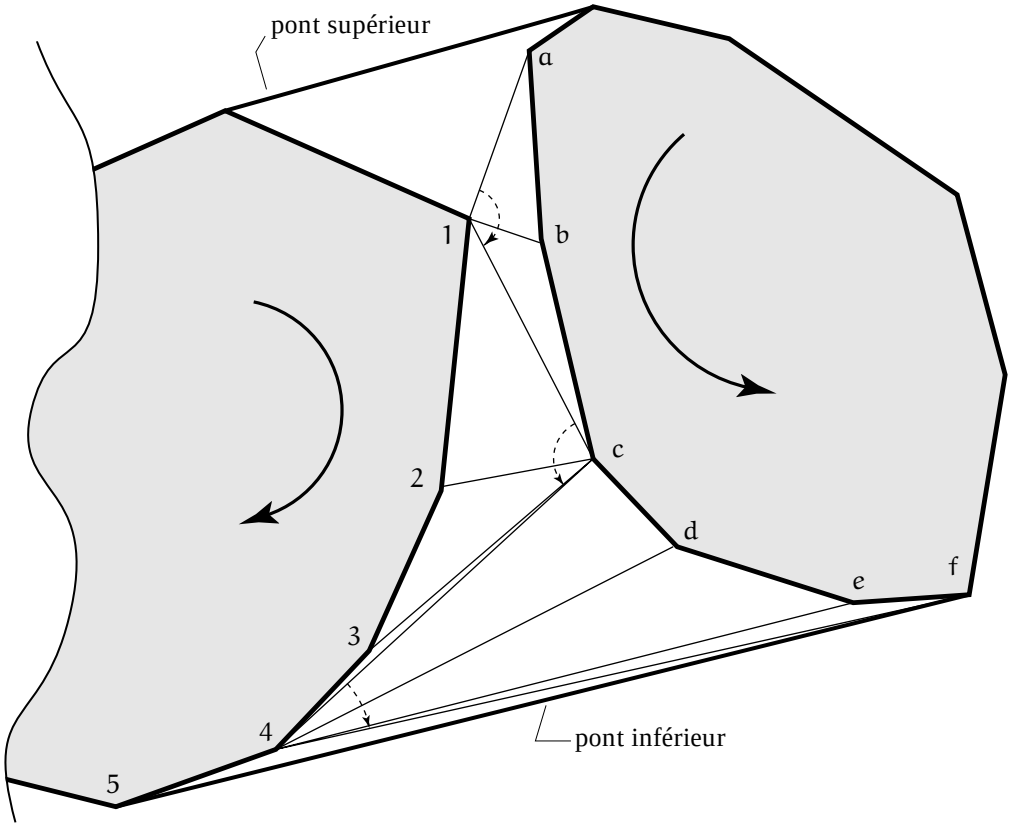


Figure 10.7: L'algorithme de Preparata et Hong

tourner. On a trouvé le pont inférieur.

Mutatis mutandis, on trouve de même le pont supérieur.

On aura noté que cette fusion se fait bien en $O(n_1 + n_2)$, ce qui garantit que l'algorithme entier tourne en $O(n \log n)$.

10.4 Implémentation en Caml

10.4.1 Le tri-fusion

Nous rappelons brièvement l'implémentation du tri-fusion, qui a l'immense avantage sur le tri dit rapide d'être beaucoup plus simple à écrire, et qui en outre profite au maximum d'un ordre partiel, alors que c'est la configuration la pire pour le tri rapide. De toutes façons, il s'agit d'un algorithme en $O(n \log n)$.

Programme 10.1 Le tri-fusion

```
1 let rec découpe_en_deux = function
2       [] -> [],[]
3     | [a] -> [a],[]
4     | a :: b :: q ->    let l1,l2 = découpe_en_deux q
5                         in (a :: l1),(b :: l2)
6 and fusionne inf l1 l2 = match l1,l2 with
7       [],_ -> l2
8     | _,[] -> l1
9     | a :: q1 , b :: q2 ->  if (inf a b) then a :: (fusionne inf q1 l2)
10                            else b :: (fusionne inf l1 q2)
11 and tri_fusion inf = function
12      [] -> []
13    | [a] -> [a]
14    | l ->  let l1,l2 = découpe_en_deux l
15            in fusionne inf (tri_fusion inf l1) (tri_fusion inf l2);;
```

La fonction `tri_fusion` prend en argument la relation d'ordre $\prec$ et la liste à trier. Dans la suite, nous supposerons ce tri compilé et accessible par les habituels `#open` et `load_object`.

10.4.2 Quelques fonctions utilitaires

Nous commençons par les quelques fonctions qui nous serons utiles dans la suite (voir le programme 10.2).

On représentera un point par un couple d'entiers. On définit simplement les fonctions qui donnent abscisse et ordonnée d'un point, ainsi que le déterminant de deux vecteurs du plan, ce qui permet d'écrire les fonctions `saillant`, `rentrant`, et, en prime, `alignés`.

10.4.3 La recherche des points extrémaux d'un polygone convexe

Dans toute la suite, nous représentons un polygone convexe par le cycle de ses sommets, pris dans l'ordre trigonométrique. Pour parler d'un point particulier du polygone, on fournira le plus souvent le cycle complet, mais après le nombre de rotations nécessaire pour que le sommet qui nous intéresse se présente le premier dans le cycle. Bref, plutôt que de donner un point, on donne un pointeur sur ce point dans le cycle.

Dans le programme 10.3, on s'intéresse à la recherche des points extrémaux d et g des deux polygones P_1 et P_2, ainsi qu'on l'a vu dans l'algorithme décrit ci-avant. d est le point le plus à droite du polygone de gauche (P_1), et g le point le plus à gauche du polygone de droite (P_2).

Les lignes 19–31 permettent de trouver le point d. Les fonctions `xici`, `xav`, `xap` renvoient les abscisses du sommet visé du polygone, de son successeur et de son prédécesseur dans

Programme 10.2 Recherche de l'enveloppe convexe : utilitaires

```
load_object "Cycles";;
#open "Cycles";;

load_object "tri_fusion";;
#open "tri_fusion";;

type point == int * int;;

let abscisse = function (px,_) -> px;;
let ordonnée = function (_,py) -> py;;

let vecteur (px,py) (qx,qy) = (qx-px,qy-py);;

let déterminant (ax,ay) (bx,by) = ax * by - ay * bx;;

let saillant p q r = 0 < (déterminant (vecteur p q) (vecteur q r));;
let rentrant p q r = 0 > (déterminant (vecteur p q) (vecteur q r));;
let alignés  p q r = 0 = (déterminant (vecteur p q) (vecteur q r));;
```

Programme 10.3 Recherche de l'enveloppe convexe : les points extrémaux

```
let point_droit p =
    let xici p = abscisse (valeur_cycle p)
    and xav  p = abscisse (valeur_cycle (avance_cycle p))
    and xap  p = abscisse (valeur_cycle (recule_cycle p))
    in
    let rec roule_avant p =
        if (xici p) < (xav p) then roule_avant (avance_cycle p)
        else p
    and roule_derrière p =
        if (xici p) < (xap p) then roule_derrière (recule_cycle p)
        else p
    in
    if (xici p) < (xav p) then roule_avant p else roule_derrière p;;

let point_gauche p =
    let xici p = abscisse (valeur_cycle p)
    and xav  p = abscisse (valeur_cycle (avance_cycle p))
    and xap  p = abscisse (valeur_cycle (recule_cycle p))
    in
    let rec roule_avant p =
        if (xici p) > (xav p) then roule_avant (avance_cycle p)
        else p
    and roule_derrière p =
        if (xici p) > (xap p) then roule_derrière (recule_cycle p)
        else p
    in
    if (xici p) > (xav p) then roule_avant p else roule_derrière p;;
```

l'ordre trigonométrique. On ira en avant ou en arrière selon que l'on est avant ou après le point cherché, choix qui est opéré en ligne 31.

Si par exemple on est avant le point d, on appelle `roule_avant`, définie dans les lignes 24–26, qui avance jusqu'à ce que l'abscisse du point se remette à décroître : c'est que d est le point d'abscisse maximale.

On obtient de façon analogue le point g.

10.4.4 L'algorithme de Preparata et Hong : les ponts inférieur et supérieur

Nous décrirons ici simplement la fonction `pont_inférieur` des lignes 46–72 du programme 10.4. On se donne les deux polygones par les cycles correspondants, en visant directement les points d et g vus précédemment.

Il va s'agir d'itérer la succession des deux rotations en appui sur P_1 et en appui sur P_2, jusqu'à ce qu'on ait trouvé le pont inférieur. À ce moment là, aucune de ces deux rotations ne sera plus possible, ce qui signalera la fin du processus.

une logique à trois états

C'est l'argument `niveau` qui permet ce contrôle. On part avec une valeur nulle (ligne 72), et on ajoute 1 à chaque fois qu'une rotation *est bloquée* (lignes 61 et 67). En revanche, dès qu'une rotation est réussie, on remet `niveau` à 0, puisque tout est à recommencer : c'est ce qui arrive aux lignes 60 et 66. Enfin, on arrête l'itération quand `niveau` vaut 2 (test en ligne 69).

Les rotations en appui à gauche ou à droite se font sans difficultés, il suffit de ne pas se tromper dans le test avec `saille` ou `rentre`, qui sont des versions modifiées de `saillant` et `rentrant` pour tenir compte du cas des points alignés.

Notons qu'ici aussi `pont_inférieur` renvoie non pas un couple de points, mais le couple des deux cycles qui représentent les polygones, en *visant* les points intéressants.

10.4.5 La fonction `enveloppeConvexe`

On termine par la fonction `enveloppeConvexe` proprement dite (voir le programme 10.5), qui prend en argument une liste de points et rend un cycle représentant le polygone convexe frontière de l'enveloppe convexe souhaitée.

On utilise une fonction récursive `enveloppeConvexeRécursif` à laquelle on fournit la liste des points triés en abscisse (ligne 138). Ainsi est-on sûr de ne pas réaliser ce tri plus d'une fois.

Intéressons-nous donc à cette fonction récursive (lignes 102 à 135).

Les lignes 103–108 évacuent les cas les plus élémentaires d'une liste de moins de 4 points, pour lesquels on fournit directement la réponse, en faisant attention à respecter le sens trigonométrique pour ranger les points d'un triangle.

Les lignes 120–125 définissent la petite et très simple fonction `coupe_en_deux` qui prend une liste et renvoie le couple de listes `(préfixe,suffixe)` formé de ses deux moitiés.

On peut ainsi utiliser la stratégie diviser pour régner en coupant en deux la liste initiale puis appelant récursivement le calcul de l'enveloppe convexe sur chacune des deux moitiés (lignes 109–113).

Il est alors temps d'utiliser ce qui précède en calculant les deux points extrémaux (ligne 114), puis le pont inférieur (ligne 116), et enfin le pont supérieur (ligne 117).

Reste à fusionner le tout. C'est la tâche de `fusionne_polygones`, qui est définie en lignes 126–135.

Le pont inférieur étant $[pq]$, on accumule à rebours les différents points du pourtour. D'abord les points de P_1 entre p et p', puis le pont supérieur $[p'q']$, et enfin les points de P_2 entre q' et q.

La fonction `tour1`, par exemple, prend en arguments la liste d'accumulation, le cycle en cours, et la balise qui indique le dernier sommet à accumuler (en l'occurrence p'). Elle s'écrit (lignes 127–129) à l'aide d'un simple appel récursif.

Pour terminer, il n'y a plus qu'à convertir la liste accumulée en un cycle. Le tour est joué.

Programme 10.4 Recherche de l'enveloppe convexe : les ponts inférieur et supérieur

```
let pont_inférieur d g =
    let saille p q r =
        let d = déterminant (vecteur p q) (vecteur q r)
        in
        (d > 0) or ( (d = 0) & ((ordonnée p) > (ordonnée r)) )
    and rentre p q r =
        let d = déterminant (vecteur p q) (vecteur q r)
        in
        (d < 0) or ( (d = 0) & ((ordonnée p) > (ordonnée r)) )
    in
    let rec tourne_en_appui_gauche (p1,p2,niveau) =
        if saille   (valeur_cycle p2)
                    (valeur_cycle p1)
                    (valeur_cycle (avance_cycle p2))
        then tourne_en_appui_gauche (p1,(avance_cycle p2),0)
        else (p1,p2,niveau + 1)
    and tourne_en_appui_droit (p1,p2,niveau) =
        if rentre   (valeur_cycle p1)
                    (valeur_cycle p2)
                    (valeur_cycle (recule_cycle p1))
        then tourne_en_appui_droit ((recule_cycle p1),p2,0)
        else (p1,p2,niveau + 1)
    and itère (p1,p2,niveau) =
        if niveau < 2 then
            itère (tourne_en_appui_droit (tourne_en_appui_gauche (p1,p2,niveau)))
        else p1,p2
    in itère (d,g,0);;

let pont_supérieur d g =
    let saille p q r =
        let d = déterminant (vecteur p q) (vecteur q r)
        in
        (d > 0) or ( (d = 0) & ((ordonnée p) < (ordonnée r)) )
    and rentre p q r =
        let d = déterminant (vecteur p q) (vecteur q r)
        in
        (d < 0) or ( (d = 0) & ((ordonnée p) < (ordonnée r)) )
    in
    let rec tourne_en_appui_gauche (p1,p2,niveau) =
        if rentre   (valeur_cycle p2)
                    (valeur_cycle p1)
                    (valeur_cycle (recule_cycle p2))
        then tourne_en_appui_gauche (p1,(recule_cycle p2),0)
        else (p1,p2,niveau + 1)
    and tourne_en_appui_droit (p1,p2,niveau) =
        if saille   (valeur_cycle p1)
                    (valeur_cycle p2)
                    (valeur_cycle (avance_cycle p1))
        then tourne_en_appui_droit ((avance_cycle p1),p2,0)
        else (p1,p2,niveau + 1)
    and itère (p1,p2,niveau) =
        if niveau < 2 then
            itère (tourne_en_appui_droit (tourne_en_appui_gauche (p1,p2,niveau)))
        else p1,p2
    in itère (d,g,0);;
```

Programme 10.5 Recherche de l'enveloppe convexe : l'algorithme proprement dit

```
let enveloppeConvexe l =
    let rec enveloppeConvexeRécursif l = match l with
          [] -> liste_en_cycle l
        | [p] -> liste_en_cycle l
        | [p;q] -> liste_en_cycle l
        | [p;q;r] -> if saillant p q r
                        then liste_en_cycle [p;q;r]
                        else liste_en_cycle [p;r;q]
        | _ ->  let l1,l2 = coupe_en_deux l
                in
                let p1,p2 = (enveloppeConvexeRécursif l1),
                            (enveloppeConvexeRécursif l2)
                in
                let d,g = (point_droit p1),(point_gauche p2)
                in
                let p,q = pont_inférieur d g
                and p',q' = pont_supérieur d g
                in
                fusion_polygones p q p' q'
    and coupe_en_deux l =
        let rec coupure_récursive i l =
            if i <= 0 then [],l
            else    let l1,l2 = coupure_récursive (i-1) (tl l)
                    in ((hd l) :: l1) , l2
        in coupure_récursive ((list_length l) / 2) l
    and fusion_polygones p q p' q' =
        let rec tour1 accu c val_p' =
            if (valeur_cycle c) = val_p' then val_p' :: accu
            else tour1 ((valeur_cycle c) :: accu) (recule_cycle c) val_p'
        and tour2 accu c val_q =
            if (valeur_cycle c) = val_q then val_q :: accu
            else tour2 ((valeur_cycle c) :: accu) (recule_cycle c) val_q
        in
        liste_en_cycle (tour2   (tour1  [] p (valeur_cycle p'))
                                q' (valeur_cycle q))
    in
    enveloppeConvexeRécursif
        (tri_fusion (fun (px,_) (qx,_) -> px < qx) l);;
```

Chapitre 11

Problèmes de proximité dans le plan

Dans ce chapitre, nous étudions quelques problèmes autour de la notion de proximité dans le plan euclidien, où nous considérons un ensemble S de n points.

11.1 Quelques problèmes classiques

11.1.1 La paire la plus rapprochée

Un des premiers problèmes qu'on peut être amené à résoudre est le problème de la paire la plus rapprochée : il s'agit de déterminer ceux des points qui sont les plus proches. Plus précisément, il s'agit de déterminer une paire $\{v, w\}$ de points distincts de S tels que

$$d(v, w) = \min\{d(u_1, u_2) \ / \ u_1 \neq u_2, u_1, u_2 \in S\}.$$

Une application est la suivante : pour un aiguilleur du ciel, rechercher les deux avions les plus proches et qui risquent donc le plus d'entrer en collision.

Nous nous intéresserons en détail à ce problème dans la section suivante.

11.1.2 Les plus proches voisins

Cette fois on dira que, si a et b sont deux points distincts de S, b est un plus proche voisin de a ce que nous noterons $a \to b$ si

$$d(a, b) = \min_{c \in S \setminus \{a\}} d(a, c).$$

Notons qu'il n'y a bien sûr pas unicité d'un plus proche voisin d'un point donné. Remarquons encore que $a \to b$ n'implique pas $b \to a$. D'ailleurs Pielou démontre en 1977 que la probabilité qu'une paire de deux points distincts a et b du plan vérifie à la fois $a \to b$ et $b \to a$ est égale à

$$\frac{6\pi}{8\pi + 3\sqrt{3}} \approx 0{,}6215.$$

La figure 11.1, page suivante, montre le graphe de la relation sur un exemple.

11.1.3 Triangulation

Il s'agit maintenant de partitionner l'enveloppe convexe des points considérés en triangles. C'est ce que pratiquent régulièrement les géomètres.

Nous retrouverons ce problème en liaison avec les diagrammes de Voronoï.

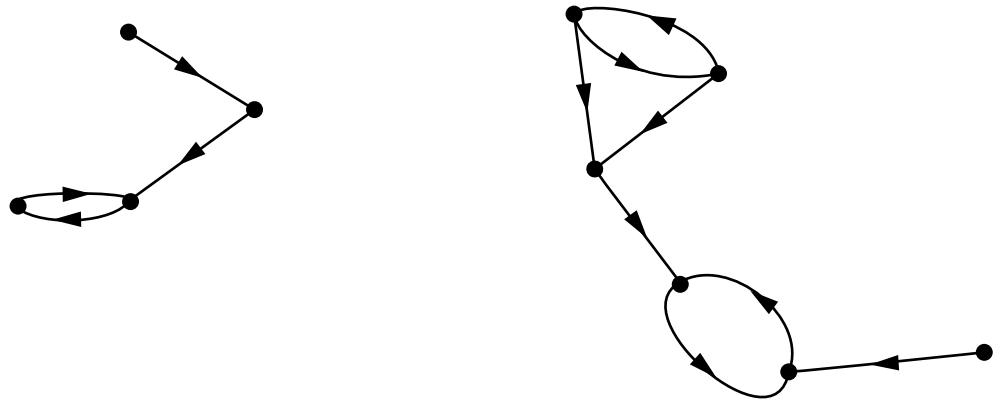


Figure 11.1: La relation $\rightarrow$ de plus proche voisinage

11.2 La paire la plus rapprochée

11.2.1 Première analyse

Nous allons nous attaquer au problème de la paire la plus rapprochée à l'aide de notre stratégie favorite : *diviser pour régner*. Soit $T(n)$ le coût de notre algorithme pour un ensemble S de n points. Nous partageons S en deux sous-ensembles S_1 et S_2, sur lesquels nous appliquons notre algorithme de façon récursive. Voilà qui coûte déjà $2T(n/2)$. Il reste à résoudre le problème pour S grâce à l'information que fournissent les résolutions pour S_1 et S_2, ce qui doit être fait en $O(n)$ si nous visons un coût global en $O(n \log n)$. C'est bien là toute la difficulté.

l'algorithme naïf tourne lui en $O(n^2)$

Supposons en effet que $\{p_1, p_2\}$ (*resp.* $\{q_1, q_2\}$) soit la paire la plus rapprochée de S_1 (*resp.* S_2). La paire la plus proche est alors ou bien $\{p_1, p_2\}$, ou bien $\{q_1, q_2\}$, ou bien encore $\{p_3, q_3\}$ où $p_3 \in S_1$ et $q_3 \in S_2$. Il faut donc *a priori* $O((n/2)^2) = O(n^2)$ pour terminer, en testant les p_3 et les q_3. Cela nous conduirait à un coût global en $O(n^2 \log n)$, ce qui fait trop.

11.2.2 Un cas particulier : la dimension 1

Nous allons nous intéresser maintenant au cas particulier de la dimension 1 (tous les points sont alignés sur une droite), en espérant trouver une idée généralisable à la dimension 2.

On a donc partitionné l'ensemble S en deux parties S_1 et S_2, en coupant la droite qui les contient à une abscisse m, puis on a trouvé les paires les plus rapprochées, $\{p_1, p_2\}$ et $\{q_1, q_2\}$, de S_1 et S_2 (cf. figure11.2).

on confond ici points et abscisses

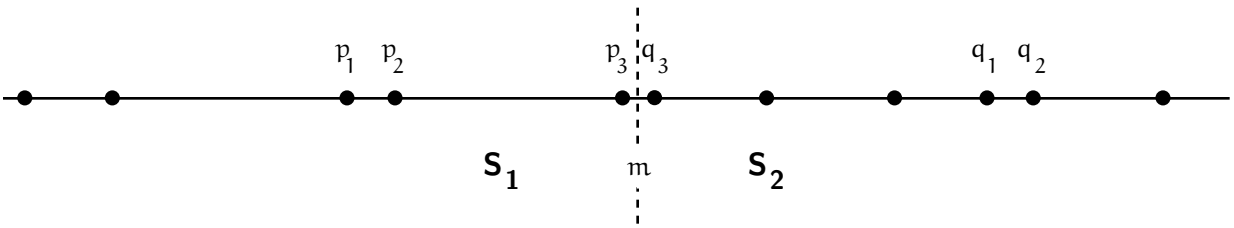


Figure 11.2: Diviser pour régner sur une droite

Posons $\delta_1 = d(p_1, p_2)$, $\delta_2 = d(q_1, q_2)$, et $\delta = \min\{\delta_1, \delta_2\}$. La remarque qui va nous sauver est la suivante : si $p_3 \in S_1$ et $q_3 \in S_2$ répondent au problème posé, alors nécessairement ils sont à une distance de m inférieure à δ. En outre, il est certain qu'il y a au plus un point de S_1 à une distance de m inférieure à δ car $\delta \leq \delta_1$. La même remarque peut être faite pour S_2. Ainsi la dernière étape de notre algorithme est-elle bien en $O(n)$.

11.2.3 Retour au problème plan

Considérons maintenant le cas du plan (voir figure 11.3).

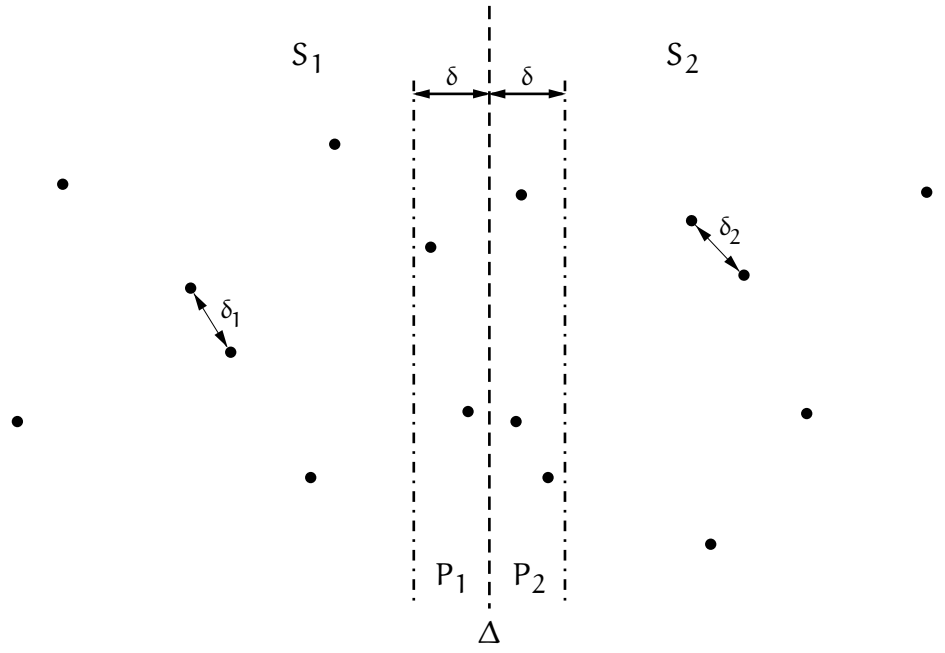


Figure 11.3: Diviser pour régner dans un plan

nous préciserons bientôt...

On partitionne l'ensemble S à l'aide d'une droite Δ qui le coupe à peu près en son milieu : S_1 est l'ensemble des points à gauche de Δ, et S_2 l'ensemble des points à droite. On applique récursivement notre procédure à chacune de ces parties, obtenant ainsi comme ci-dessus p_1 et p_2 qui réalisent dans S_1 la plus petite distance δ_1, et q_1, q_2 de distance δ_2 dans S_2. On note toujours $\delta = \min\{\delta_1, \delta_2\}$. On pourra chercher des paires qui sont plus proches et dont les éléments sont un $p_3 \in S_1$ et un $q_3 \in S_2$ en imposant que $p_3 \in P_1$ et $q_3 \in P_2$ où P_1 (*resp.* P_2) désigne l'ensemble des points de S_1 (resp. S_2) qui sont distants d'au plus δ de Δ.

Le problème est qu'on n'est plus du tout assuré de l'unicité de ces points p_3 et q_3 : il se pourrait même que $S_1 \subset P_1$ et $S_2 \subset P_2$, et on serait alors ramené à un algorithme en $O(n^2)$.

voir la figure 11.4, page suivante

En réalité, tout n'est pas si grave. En effet, si p est fixé dans P_1, il y a au plus 5 points dans P_2 distants de p de moins de δ, puisque la distance de deux points quelconques de S_2 est supérieure à δ_2, donc à δ.

la fusion après les deux appels récursifs

Cette fois, ça semble bon : nous avons en effet $n/2$ points p au maximum, et pour chacun d'entre eux au plus 5 points q, et donc cette étape de l'algorithme tourne en $O(n)$, et l'algorithme entier en $O(n \log n)$.

Sauf que...

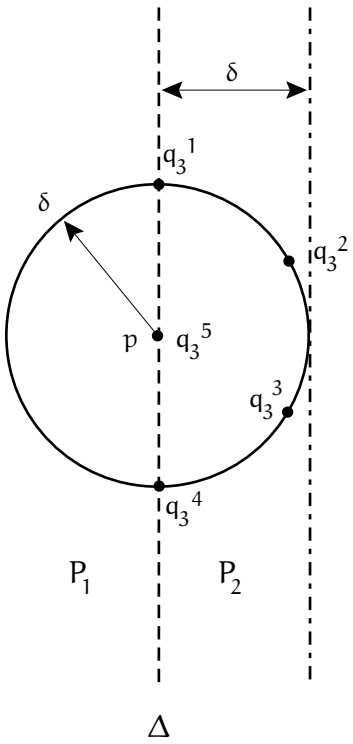


Figure 11.4: La position la plus défavorable, avec 5 points à visiter

Sauf que si nous savons que pour chaque point p il n'y a que 5 points au plus dans P_2 à considérer, nous ne savons pas pour autant les trouver !

L'idée va consister à trier les points de P_2 par ordre croissant des abscisses de leurs projections sur Δ. Pour simplifier nous séparerons les points de S par une droite Δ parallèle à l'axe des y. Si Δ a pour équation $x = m$, P_2 est l'ensemble des points tels que $m \leq x \leq x + \delta$. Nous les trions grâce à l'ordre sur leurs ordonnées. Si p a pour ordonnées y, nous trouverons en un temps constant les points q de P_2 tels que $m \leq x_q \leq m + \delta$ et $y - \delta \leq y_q \leq y + \delta$. *pourquoi pas ?*

Mais au fait : trier les points de P_2, qui sont au nombre de $n/2$ éventuellement, a un coût en $O(n \log n)$ ... nous sommes à nouveau piégés, notre algorithme tourne en $O(n \log^2 n)$!

Ce qui nous sauve encore ici est qu'on peut faire le tri une bonne fois pour toutes avant tout appel récursif et là, ouf, ça marche.

Écrivons donc informellement l'algorithme obtenu :

1. trier l'ensemble des points de S par abscisses et par ordonnées ;
2. partager S en S_1 et S_2 grâce au tri en abscisses croissantes effectué, et retenir l'abscisse médiane m de séparation ;
3. chercher par un appel récursif — sans l'étape 1 qui n'est effectuée qu'une fois pour toutes — les paires (p_1, p_2) et (q_1, q_2) les plus rapprochées ;
4. poser $\delta = \min\{d(p_1, p_2), d(q_1, q_2)\}$;
5. pour tous les points p tels que $m - \delta \leq x_p$, et par ordre d'ordonnées croissantes, examiner les points q tels que $m \leq x_q \leq m + \delta$ et $|y_q - y_p| \leq \delta$ pour retenir d'éventuels couples (p, q) qui optimiseraient la distance.

Cet algorithme tourne bien en $O(n \log n)$ d'après l'étude que nous en avons faite.

11.2.4 Implémentation en Caml

Quelques utilitaires simples

Programme 11.1 Paire la plus proche : quelques utilitaires

```
1  load_object "Cycles";;
2  #open "Cycles";;
3
4  type point == int * int;;
5  type point_ou_bord = Point of point | Bord;;
6
7  let rec découpe_en_deux = function
8        [] -> [],[]
9      | [a] -> [a],[]
10     | a :: b :: q ->    let l1,l2 = découpe_en_deux q
11                         in (a :: l1),(b :: l2)
12 and fusionne inf l1 l2 = match l1,l2 with
13       [],_ -> l2
14     | _,[] -> l1
15     | a :: q1 , b :: q2 ->  if (inf a b) then a :: (fusionne inf q1 l2)
16                            else b :: (fusionne inf l1 q2)
17 and tri_fusion inf = function
18       [] -> []
19     | [a] -> [a]
20     | l ->  let l1,l2 = découpe_en_deux l
21             in fusionne inf (tri_fusion inf l1) (tri_fusion inf l2);;
22
23 let carré n = n * n;;
24
25 let distance (x,y) (x',y') = carré(x-x') + carré(y-y');;
26
27 let numérote l =
28     let rec numérote_rec n = function
29           [] -> []
30         | a :: q -> (n,a) :: (numérote_rec (n+1) q)
31     in
32     numérote_rec 0 l;;
33
34 let rec filtre prédicat = function
35       [] -> []
36     | a :: q -> if (prédicat a) then a :: (filtre prédicat q)
37                 else (filtre prédicat q);;
38
39 let est_vide = function [] -> true | _ -> false;;
```

Caml préfère les entiers, nous aussi

On définit ici le type `point`, qui représente tout simplement un couple d'entiers, et le type `point_ou_bord` qui sera utile tout à l'heure quand nous utiliserons des cycles d'objets de ce type. En effet, nous voudrons pouvoir avancer comme reculer dans la liste des points, mais sans dépasser la tête ou la queue, bref, en balisant le bord de la liste.

le tri-fusion est tellement plus simple à écrire !

On retrouve le tri-fusion qu'on a déjà vu, en lignes 7–21.

Viennent ensuite quelques fonctions très simples, comme `carré` (ligne 23), `distance` (ligne 25), ou `est_vide` (ligne 39). On a écrit également la classique fonctionnelle `filtre` qui ne conserve de sa liste argument que ceux de ses éléments qui satisfont au prédicat fourni (lignes 34–37).

Enfin, on trouvera en lignes 27–32 la fonction `numérote` qui transforme une liste $[a; b; c; d]$ par exemple en $[(0, a); (1, b); (2, c); (3, d)]$, et qui a le type `'a list -> (int * 'a) list`.

Une programmation astucieuse

Programme 11.2 Paire la plus proche : coupure d'une liste

```
40 let coupe_au_milieu l =
41     let rec avance l1 a1 = function
42           [] -> a1,l1
43         | l2 -> let a,l =
44                     if est_vide (tl l2)
45                         then avance (tl l1) a1 []
46                         else avance (tl l1) a1 (tl (tl l2))
47                 in ((hd l1) :: a),l
48     in avance l [] l;;
```

prouve cet algorithme !

On donne ici une programmation astucieuse qui permet de couper en deux une liste : après `let l1,l2 = coupure_au_milieu l;;`, on peut affirmer que `l` est identique à `l1 @ l2`, et que `l1` a la même taille (à une unité près) que `l2`.

On pourrait bien sûr écrire plus simplement

```
let coupure_au_milieu l =
    let rec coupure_récursive i l =
        if i <= 0 then [],l
        else    let l1,l2 = coupure_récursive (i-1) (tl l)
                in ((hd l) :: l1) , l2
    in coupure_récursive ((list_length l) / 2) l
```

mais cela demande deux parcours complets de la liste `l`, un pour la procédure récursive, et un pour évaluer `list_length l`.

On vérifiera que la fonction proposée ne fait qu'un passage dans la liste argument.

La fusion

`cherche_mieux` réalise l'essentiel de la fusion : il attend en argument les deux ensembles P_1 et P_2 évoqués dans la description de notre algorithme, sous la forme de cycles de points (pour pouvoir aller en avant comme en arrière) balisés grâce au constructeur `Bord`, ainsi que la paire la plus proche déjà trouvée, constituée des deux points a et b à distance δ.

En lignes 66–69, on gère les cas d'arrêt.

En lignes 70–71, on s'occupe du cas où le point courant de P_2 serait trop bas : il convient alors de s'appeler récursivement avec le même point de P_1, mais le point suivant de P_2.

Sinon, il faut tester les 5 points à venir dans P_2 (voir la discussion de l'algorithme) : c'est le but des appels imbriqués à la procédure `essai`, en ligne 74, appels qu'on englobe dans un `try` (lignes 73–78) pour s'arrêter dès qu'on envisage un point de P_2 trop haut.

La procédure `essai` est définie en lignes 54–64. Elle évacue rapidement en lignes 57–60 le cas où le point courant de P_2 est trop haut en déclenchant une erreur (la définition de l'exception est en lignes 49–51).

Sinon, elle compare les distances, en lignes 61–64.

Programme 11.3 Paire la plus proche : le cœur du problème

```
49 exception Trop_haut of
50         (point*point*int*
51          point_ou_bord cycle_bien_chaîné*point_ou_bord cycle_bien_chaîné);;
52
53 let rec cherche_mieux a b delta p1 p2 =
54     let essai (a,b,delta,p1,p2) =
55         match (valeur_cycle p1),(valeur_cycle p2) with
56               Bord,_ -> raise (Trop_haut (a,b,delta,p1,p2))
57             | _,Bord -> raise (Trop_haut (a,b,delta,p1,p2))
58             | Point(x1,y1),Point(x2,y2)
59                 ->  if (y2 > y1) & (carré(y2-y1) >= delta) then
60                         raise (Trop_haut (a,b,delta,p1,p2))
61                     else    let d = distance (x1,y1) (x2,y2)
62                             in
63                             if d < delta then (x1,y1),(x2,y2),d,p1,(avance_cycle p2)
64                             else a,b,delta,p1,(avance_cycle p2)
65     in
66     match (valeur_cycle p1),(valeur_cycle p2) with
67           Bord,_ -> [a;b]
68         | _,Bord -> [a;b]
69         | Point(x1,y1),Point(x2,y2)
70             ->  if (y2 < y1) & (carré(y1-y2) >= delta) then
71                     cherche_mieux a b delta p1 (avance_cycle p2)
72                 else
73                     try let a,b,delta,_,_  =
74                                 essai(essai(essai(essai(essai(a,b,delta,p1,p2)))))
75                         in
76                         cherche_mieux a b delta (avance_cycle p1) p2
77                     with Trop_haut(a,b,delta,_,_)
78                             -> cherche_mieux a b delta (avance_cycle p1) p2;;
```

La fonction appelante

Programme 11.4 Paire la plus proche : la fonction principale

```
79 let paire_la_plus_proche l =
80     let rec paire_la_plus_proche_rec lx ly = match lx with
81       [] -> []
82     | [_,a] -> [a]
83     | [_,a;_,b] -> [a;b]
84     | [_,a;_,b;_,c]
85         ->  if (distance a b) < (distance a c)
86             then    if (distance a b) < (distance b c)
87                     then [a;b]
88                     else [b;c]
89             else    if (distance a c) < (distance b c)
90                     then [a;c]
91                     else [b;c]
92     | _ ->
93         let l1x,l2x = coupe_au_milieu lx
94         in
95         let indexMilieu = match (hd l2x) with i,(_,_) -> i
96         in
97         let l1y = filtre (function (i,_) -> i < indexMilieu) ly
98         and l2y = filtre (function (i,_) -> i >= indexMilieu) ly
99         in
100        let [a1;b1] = paire_la_plus_proche_rec l1x l1y
101        and [a2;b2] = paire_la_plus_proche_rec l2x l2y
102        in
103        let delta1,delta2 = (distance a1 b1),(distance a2 b2)
104        in
105        let m = match (hd l2x) with _,(x,_) -> x
106        and aMin,bMin,delta = if delta1<delta2  then a1,b1,delta1
107                                                else a2,b2,delta2
108        in
109        let p1 = filtre (function (_,(x,_)) -> carré(x-m) <= delta) l1y
110        and p2 = filtre (function (_,(x,_)) -> carré(x-m) <= delta) l2y
111        in
112        let p1 = avance_cycle (liste_en_cycle
113                               (Bord :: (map (function (_,a) -> Point(a)) p1)))
114        and p2 = avance_cycle (liste_en_cycle
115                               (Bord :: (map (function (_,a) -> Point(a)) p2)))
116        in
117        cherche_mieux aMin bMin delta p1 p2
118    in
119    let lx = numérote
120                (tri_fusion (fun (ax,_) (bx,_) -> ax < bx) l)
121    in
122    let ly = tri_fusion (fun (_,(_,ay)) (_,(_,by)) -> ay < by) lx
123    in
124    paire_la_plus_proche_rec lx ly;;
```

Pour s'assurer qu'on n'opère qu'une fois le tri, on l'effectue avant tout appel à la fonction récursive, en lignes 119–122, où on commence par construire la liste des points triée en x, à la numéroter (on a construit ainsi `lx`), puis à trier cette liste cette fois en y, obtenant `ly`.

La fonction récursive fait évidemment tout le travail. Elle commence par évacuer le cas d'une liste de moins de 4 points (lignes 81–91). Ensuite on opère la découpe en deux listes, en lignes 93–98 ; on notera tout particulièrement l'intérêt d'avoir numéroté les points par ordre des x croissants, sans quoi la découpe de `ly` eût été impossible. Les appels récursifs sur les deux moitiés ont lieu en lignes 100 et 101. On peut alors déterminer la paire la plus proche, c'est-à-dire les a, b et δ qu'attend `cherche_mieux` : c'est l'objet des lignes 103–107. Grâce à `filtre`, on détermine les deux ensembles P_1 et P_2, en lignes 109–115. Il n'y a plus qu'à appeler `cherche_mieux`.

11.3 Diagrammes de Voronoï

11.3.1 Définition

On considère un ensemble fini S de n points $p_0, \ldots, p_{n-1}$. Pour chaque indice i tel que $0 \leq i < n$, on cherche à déterminer l'ensemble V_i des points m du plan plus proches de m_i que des autres m_j, défini par

$$V_i = \{m \;/\; \forall j \neq i, d(m, p_i) \leq d(m, p_j)\}.$$

nos demi-plans sont tous supposés fermés, ils contiennent leur frontière

Si on considère le cas de deux points p_i et p_j, la réponse est élémentaire : il s'agit des deux demi-plans limités par la médiatrice du segment $[p_i p_j]$. Dans toute la suite nous noterons $H(p_i, p_j)$ celui de ces deux demi-plans qui contient le point p_i. Ainsi $H(p_i, p_j) \cup H(p_j, p_i)$ est le plan entier. Alors, par définition même de V_i, on dispose de

$$V_i = \bigcap_{j \neq i} H(p_i, p_j).$$

Ceci prouve que V_i est — en tant qu'intersection de demi-plans, et donc aussi de convexes — une partie convexe du plan qui contient p_i et aucun autre point de S et dont la frontière est polygonale. De deux choses l'une : ou bien V_i est compact, c'est un polygone, ou bien V_i n'est pas borné, et sa frontière est une suite de segments encadrée par deux demi-droites. Le plus parlant est de considérer la figure 11.5 qui montre la partition du plan en les V_i dans un cas particulier. Un tel diagramme est appelé diagramme de Voronoï, qui les a introduits en 1908 à l'occasion d'un traité sur les formes quadratiques.

11.3.2 Quelques propriétés

Dans toute la suite, pour éviter des cas particuliers qui n'apportent pas grand chose à la compréhension des notions introduites, nous ferons la supposition suivante :

> Il n'y a pas dans S quatre points cocycliques.

Notons que comme une arête du diagramme de Voronoï est un segment de médiatrice, elle est donc arête commune d'exactement deux polygones. En outre nous avons le

Théorème 5 *Chaque sommet du diagramme de Voronoï est intersection d'exactement trois arêtes.*

✧ Soit $a_1, a_2, \ldots, a_k$ les arêtes qui se rencontrent en un sommet v du diagramme de Voronoï, numérotées dans l'ordre de leurs angles polaires. Renumérotons les points de S de telle sorte que a_i soit un côté commun aux polygones V_{i-1} et V_i, avec la convention habituelle : a_1 est un côté commun aux polygones V_1 et V_k. Mais ces arêtes sont des médiatrices, et c'est donc dire que v est équidistant des points $p_1, p_2, \ldots, p_k$. Grâce à notre hypothèse il est alors clair que $k \leq 3$. Supposons pour terminer que $k = 2$. Alors a_1 est côté commun à V_1 et V_2, et a_2 aussi, bref ce sont deux segments contigus de la même médiatrice, et il n'y aurait pas là de sommet v du diagramme de Voronoï. Finalement on a bien $k = 3$.
✦

considérer à nouveau la figure

On peut également dire que chaque sommet du diagramme de Voronoï est le centre d'un cercle passant par trois points de S. Dans la suite, si v est un sommet du diagramme de Voronoï, nous noterons $C(v)$ le cercle correspondant. On dispose alors de l'intéressante propriété suivante.

Théorème 6 *Pour chaque sommet v du diagramme de Voronoï, le disque ouvert de frontière $C(v)$ ne contient aucun point de S.*

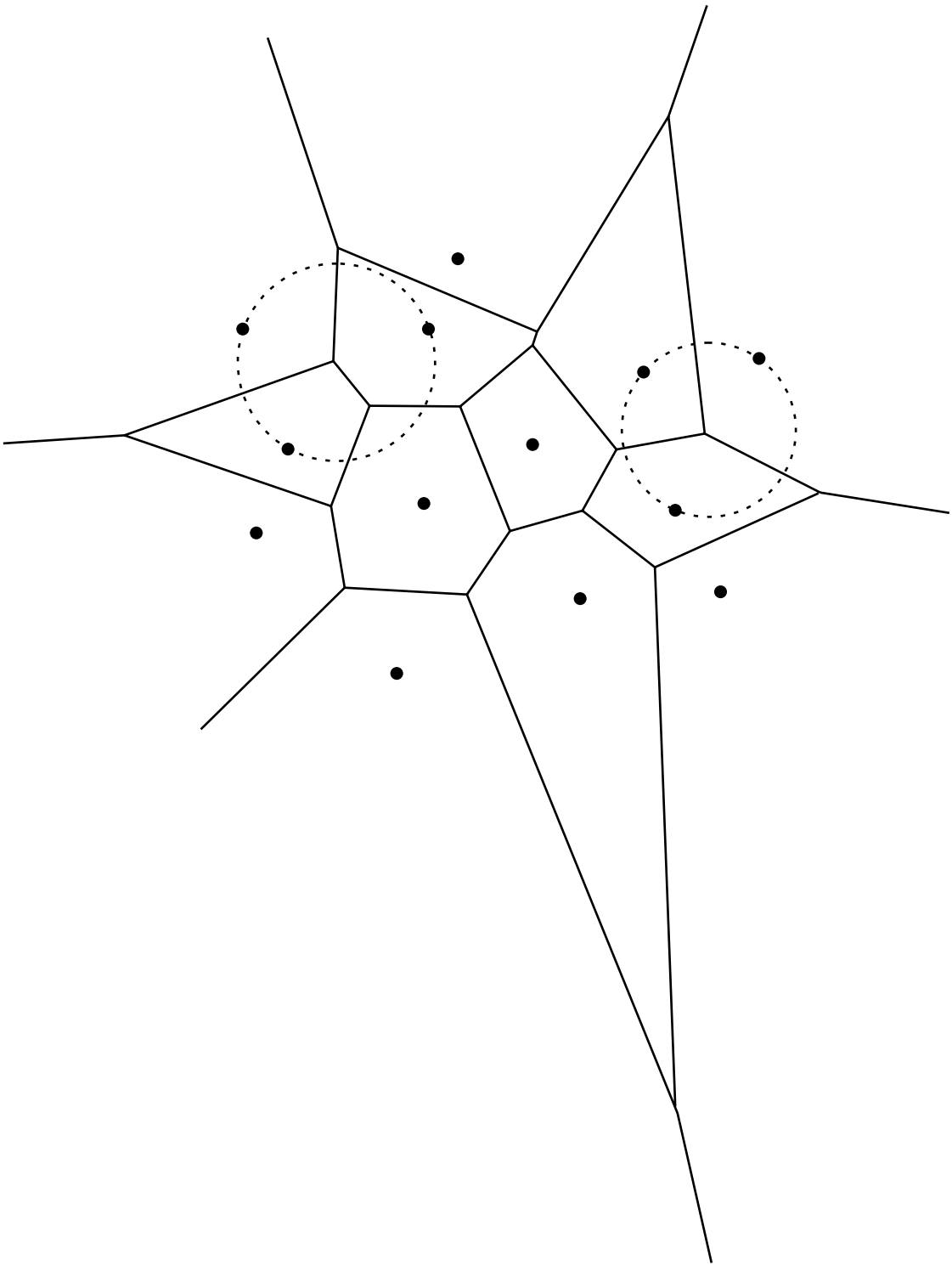


Figure 11.5: Un exemple de diagramme de Voronoï

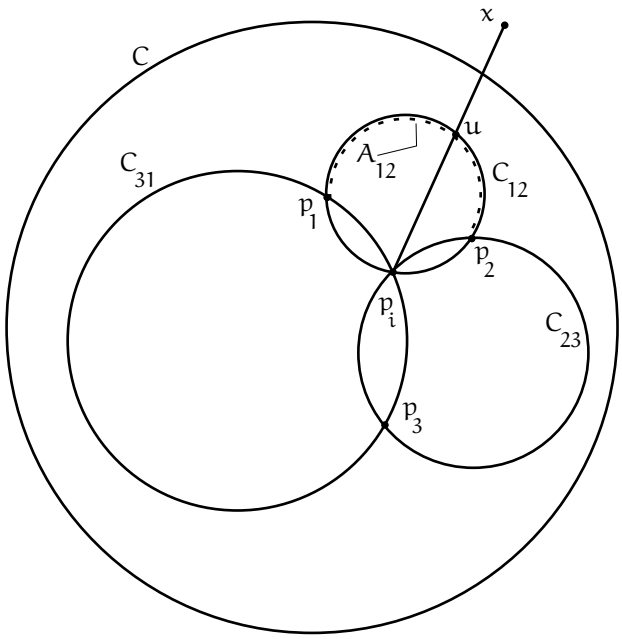


Figure 11.6: Support de la démonstration du théorème 8

✧ Raisonnons par l'absurde. Supposons que $C(v)$ passe par les trois points p_1, p_2, p_3, et que le disque ouvert contienne un quatrième point p_4. Mais alors v est plus proche de p_4 que de chacun des p_1, p_2, p_3, et donc $v \in V_4$ mais $v \notin V_1$, $v \notin V_2$ et $v \notin V_3$. Ce qui fournit notre contradiction puisqu'on a supposé que v est le sommet commun aux trois polygones V_1, V_2, V_3.
✥

Voici encore un petit

Théorème 7 *Soit p_i un point de S. Si p_j est un plus proche voisin de p_i alors le polygone V_i du diagramme de Voronoï voit une de ses arêtes portée par la médiatrice de $[p_i p_j]$.*

✧ Soit m le milieu du segment $[p_i p_j]$. Supposons un instant que m ne soit pas sur la frontière de V_i. Comme V_i est inclus dans $H(p_i, p_j)$, m est alors nécessairement à l'extérieur de V_i. Ainsi le segment $[p_i v]$ rencontre une arête de V_i en un certain point u, et $d(p_i, u) < d(p_i, m)$. u est sur une arête de V_i donc sur la médiatrice d'un certain segment $[p_i p_k]$. Mais alors $d(p_i, u) = \frac{1}{2} d(p_i, p_k) < d(p_i, m) = \frac{1}{2} d(p_i, p_j)$, et donc p_k serait plus près de p_i que p_j, ce qui est exclu par l'hypothèse.
✥

Voici ensuite un résultat qui permet de retrouver l'enveloppe convexe de S connaissant son diagramme de Voronoï :

Théorème 8 *Le polygone V_i n'est pas borné si et seulement si p_i est sur la frontière de l'enveloppe convexe de S.*

✧ On pourra observer la figure 11.6 comme support de cette démonstration.
Pour cette démonstration, nous utiliserons le lemme qui dit qu'un point p n'est pas un sommet de l'enveloppe convexe de S si et seulement si il est à l'intérieur d'un triangle rst de points de S.
Si donc p_i n'est pas sur la frontière de l'enveloppe convexe de S, c'est qu'il est à l'intérieur d'un triangle $p_1 p_2 p_3$ où p_1, p_2 et p_3 sont trois points de S. On considère alors les cercles

C_{12}, C_{23} et C_{31} respectivement circonscrits aux triangles $p_i p_1 p_2$, $p_i p_2 p_3$ et $p_i p_3 p_1$. Sur C_{12} on considère plus particulièrement l'arc A_{12} qui est limité par p_1 et p_2 et qui ne contient pas p_i. Remarquons que si m est un point de cet arc, m est plus proche de ou bien p_1 ou bien p_2 (ou bien des deux) que de p_i. On a de même des arcs A_{23} et A_{31} avec les propriétés analogues. Soit enfin C un cercle tel que le disque qu'il délimite contienne l'intégralité des trois cercles précédents.
Nous allons montrer que tout point x extérieur à ce disque est plus proche de l'un ou l'autre des points p_1, p_2 ou p_3 que de p_i. Ceci prouvera que V_i est tout entier inclus dans ce disque de frontière C, et partant, qu'il est borné.
En effet, le segment $[p_i x]$ coupe l'un des côtés du triangle $p_1 p_2 p_3$, par exemple $p_1 p_2$. Mais alors il coupe également l'arc A_{12}, en un point u. On a déjà dit que u est plus proche de p_1 ou p_2 que de p_i, et c'est gagné.
Réciproquement, supposons que V_i soit compact, et montrons que p_i n'est pas sur la frontière de l'enveloppe convexe de S. En effet, V_i a pour frontière une suite de segments $\sigma_1, \sigma_2, \ldots, \sigma_k$ (avec $k \geq 3$). Chaque σ_j est porté par la médiatrice d'un certain $p_i p'_j$, où $p'_j \in S$. Mais alors il est clair que p_i est à l'intérieur du polygone de sommets p'_1, $p'_2, \ldots, p'_k$, et donc pas sur la frontière de l'enveloppe convexe de S.
◆

11.3.3 Applications

Dual du diagramme de Voronoï : la triangulation de Delaunay

En 1934, Delaunay propose de considérer le graphe plan obtenu en reliant par un segment de droite chaque paire de points p_i, p_j de S tels que les polygones V_i et V_j du diagramme de Voronoï partagent une arête commune (qui est donc alors contenue dans la médiatrice du segment). Remarquons qu'il n'y a pas de raison pour que le segment de Delaunay coupe la médiatrice en question. On trouvera la triangulation de Delaunay associée au diagramme de Voronoï déjà tracé dans la figure 11.7, page suivante (le diagramme de Voronoï est en pointillés, la triangulation de Delaunay en trait épais). Il s'agit là d'un résultat lié à la dualité : le dual du théorème 5 sur le diagramme de Voronoï dit exactement que la manipulation précédente conduit effectivement à une *triangulation*, c'est-à-dire à une partition de l'enveloppe convexe de S en triangles (à *trois côtés*, comme il y avait *trois arêtes* issues de chaque sommet du diagramme de Voronoï).

Autres applications

On a déjà dit comment la donnée du diagramme de Voronoï permet de reconstituer l'enveloppe convexe (voir le théorème 8).

D'autre part, depuis le théorème 7, on sait que si p_j est un plus proche voisin de p_i, alors la médiatrice de $[p_i p_j]$ porte une arête du diagramme de Voronoï. Ainsi, inversement, étant donné p_i, on cherchera ses plus proches voisins en considérant successivement les seuls p_j tels que V_i et V_j aient une arête commune.

En fait, si nous savions construire en $O(n \log n)$ le diagramme de Voronoï, les remarques précédentes permettraient de résoudre le problème de l'enveloppe convexe et celui des plus proches voisins pour un coût du même ordre.

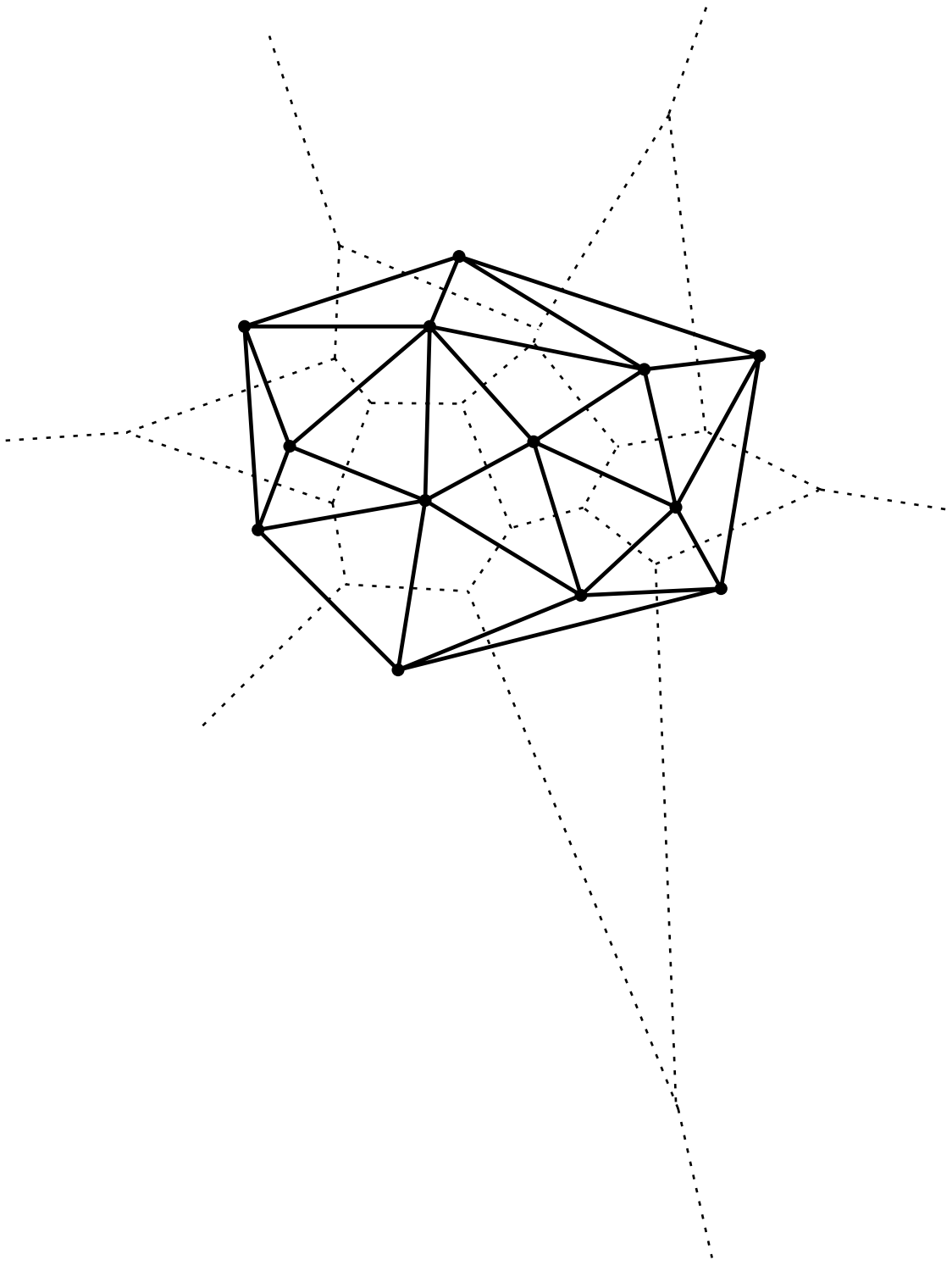


Figure 11.7: La triangulation de Delaunay

11.4 Algorithme de Tsai

11.4.1 Quelques préliminaires

cet algorithme a été publié en juin 1993

Nous présentons ici un algorithme proposé par V. Tsai pour construire la triangulation de Delaunay d'un ensemble de n points du plan. Tsai annonce que son algorithme (pour ses deux dernières étapes, du moins) est *à peu près linéaire en* n...

Nous nous abstiendrons de toute justification complète de l'algorithme, ce qui dépasserait largement le cadre de ce poly. À ce jour, l'évaluation rigoureuse de son efficacité reste à faire.

Voici tout d'abord deux résultats autour de la triangulation de Delaunay qui nous serviront dans la suite.

Théorème 9 (Critère de Delaunay) *Le cercle circonscrit à un triangle* pqr *de la triangulation de Delaunay d'un ensemble de* n *points du plan ne contient aucun autre point de l'ensemble.*

C'est en fait une reformulation du théorème 6.

Voici encore, sans démonstration, un autre critère, dû à Lawson, en 1972 :

Théorème 10 (Critère du max-min) *Soit* $pqrs$ *les sommets d'un quadrilatère convexe constitué de deux triangles* pqs *et* qrs *de la triangulation de Delaunay d'un ensemble de* n *points du plan qui partagent une arête commune* qs. *Soit* α *le plus petit des six angles aux sommets des deux triangles. Considérons l'autre diagonale du quadrilatère, ou encore les triangles* pqr *et* rsp, *et soit* β *le plus petit des six angles aux sommets de ces deux nouveaux triangles. Alors* $\alpha \geq \beta$.

On peut dire que la triangulation de Delaunay maximise le min des six angles en question ; voir la figure 11.8 qui illustre les deux critères à la fois.

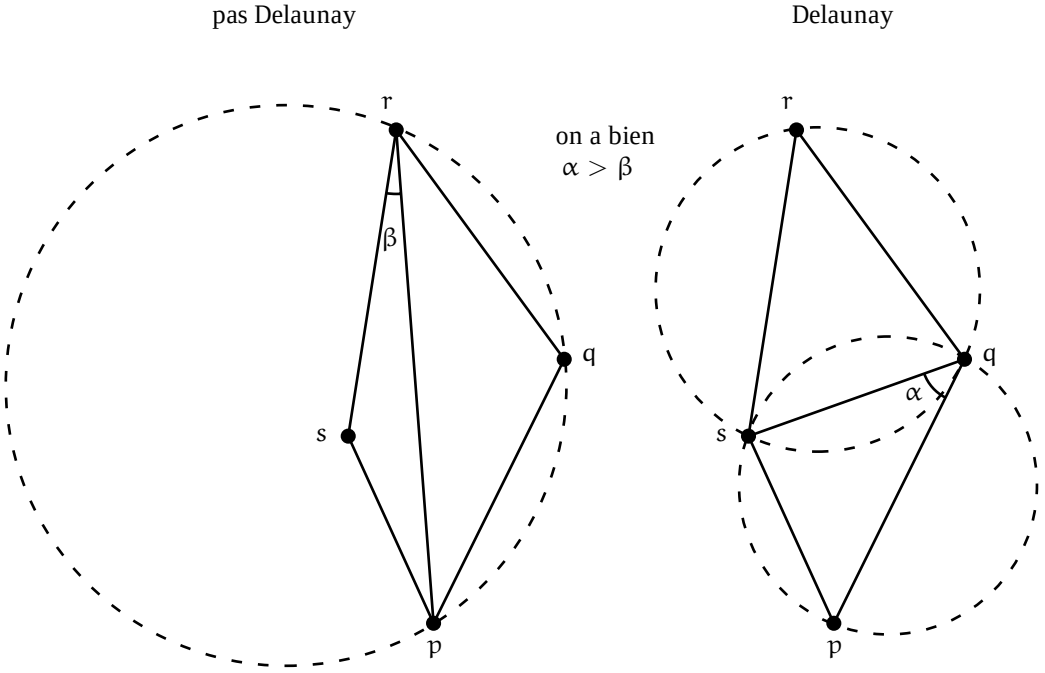


Figure 11.8: Illustration du critère du max-min

11.4.2 Première étape

La première chose à faire est de déterminer l'enveloppe convexe des n points du plan considérés. Nous avons déjà traité ce problème dans le chapitre précédent. Nous n'y revenons donc pas davantage.

11.4.3 Deuxième étape

Il s'agit maintenant de construire la triangulation de Delaunay des seuls p points de l'enveloppe convexe qu'on vient de déterminer. Notons-les a_0, a_1, ..., a_{p-1}, en les numérotant dans le sens trigonométrique : les arêtes du polygone convexe considéré sont les $a_0a_1, \ldots, a_{p-2}a_{p-1}$, $a_{p-1}a_0$.

Voici alors l'algorithme proposé par Tsai pour trouver les triangles de la triangulation de Delaunay de l'enveloppe convexe.

étape 0. on définit un ensemble d'arêtes $\mathcal{A}$ qu'on initialise avec la liste des arêtes du polygone, c'est-à-dire les a_0a_1, ..., $a_{p-2}a_{p-1}$, $a_{p-1}a_0$, ainsi qu'un ensemble $\mathcal{T}$ de triangles initialement vide ;

étape 1. si $\mathcal{A}$ est vide, on a fini : $\mathcal{T}$ contient la liste des triangles de la triangulation de Delaunay cherchée, STOP ; sinon, continuer à l'étape suivante ;

étape 2. on choisit une arête ab de $\mathcal{A}$, qu'on retire de cet ensemble ;

étape 3. on passe en revue les points c du nuage jusqu'à trouver un triangle abc qui ne soit pas déjà dans $\mathcal{T}$ et dont le disque limité par son cercle circonscrit ne contienne aucun autre point du nuage (c'est le critère de Delaunay) ;

étape 4. on ajoute à $\mathcal{T}$ le triangle abc ainsi trouvé ;

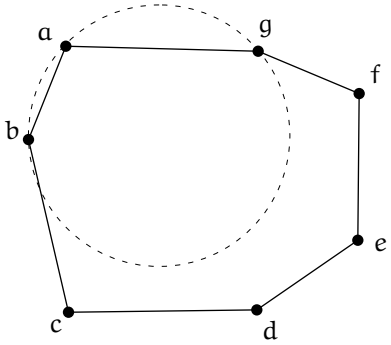
surprenant, non?

étape 5. si l'arête bc était dans $\mathcal{A}$, on la supprime, sinon on l'ajoute ; on fait de même pour l'arête ac : on la supprime si elle se trouve déjà dans $\mathcal{A}$, sinon on l'ajoute ;

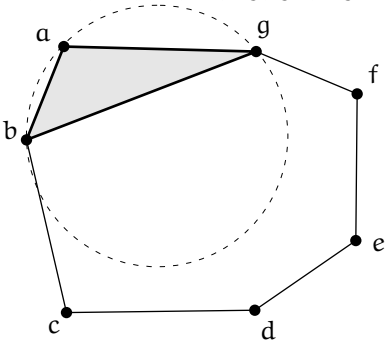
étape 6. on retourne à l'étape 1.

On trouvera un exemple en figure 11.9, page suivante.

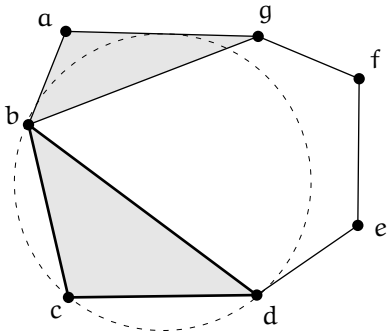
❶ $\mathcal{T} = \emptyset$
$\mathcal{A} = ab, bc, cd, de, ef, fg, ga$



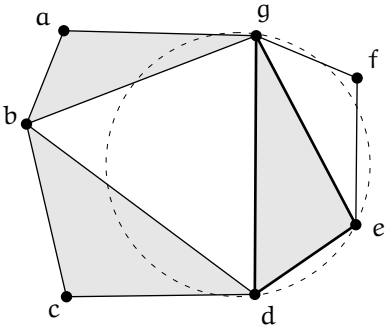
❷ $\mathcal{T} = abg$
$\mathcal{A} = bc, cd, de, ef, fg, ga, bg$



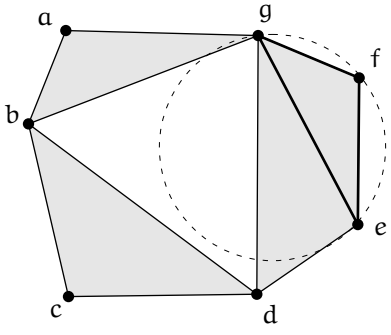
❸ $\mathcal{T} = abg, bcd$
$\mathcal{A} = de, ef, fg, bg, bd$



❹ $\mathcal{T} = abg, bcd, deg$
$\mathcal{A} = ef, fg, bg, bd, eg, dg$



❺ $\mathcal{T} = abg, bcd, deg, efg$
$\mathcal{A} = bg, bd, dg$



❻ $\mathcal{T} = abg, bcd, deg, efg, bdg$
$\mathcal{A} = \emptyset$

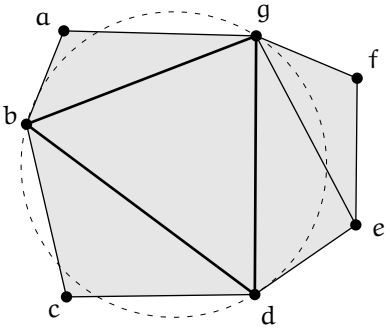


Figure 11.9: Deuxième étape de l'algorithme de Tsai

11.4.4 Troisième étape

Nous allons ajouter un à un chacun des $n - p$ points que nous n'avons pas encore considérés.

on procède ainsi : on trouve un premier triangle, puis on essaie ceux qui ont avec lui au moins un côté commun, et on itère. . .

Pour ajouter un point m (et mettre à jour la triangulation de Delaunay, bien sûr), nous commençons par déterminer l'ensemble des triangles abc de la triangulation courante qui chapeautent m : on dira qu'un triangle *chapeaute* un point si ce point est à l'intérieur du cercle circonscrit au triangle. La réunion de ces triangles constitue ce qu'on peut appeler la zone d'influence du point m (voir la figure 11.10, où la zone d'influence est grisée).

On supprime alors de la triangulation tous les côtés communs à deux triangles à l'intérieur de la zone d'influence. On termine en ajoutant tous les segments qui relient m à un des sommets à la frontière de sa zone d'influence.

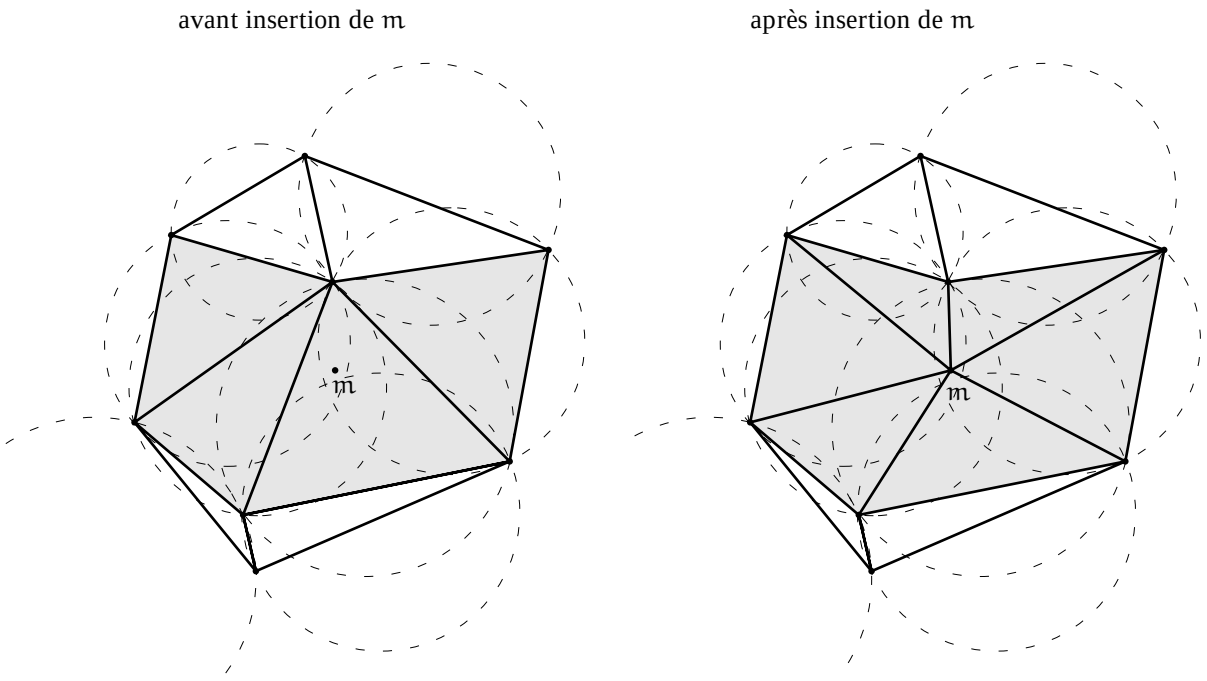


Figure 11.10: La dernière étape de l'algorithme de Tsai

Après avoir ajouté ainsi tous les points manquants, on obtient la triangulation de Delaunay cherchée.

11.4.5 Remarques

c'est déjà bien assez compliqué comme ça !

Tsai propose en réalité quelques raffinements que nous ne présentons pas ici, et qui ont pour but d'accélérer les différentes étapes de l'algorithme qu'il expose. En particulier, dans l'étape 2, il montre comment trouver rapidement le prochain triangle à ajouter à la liste. De même, pour la phase finale, il explique comment accélérer la détermination de la zone d'influence d'un point. Nous n'avons même pas utilisé la remarque marginale faite à l'occasion dans l'implémentation que nous proposons ci-après.

11.4.6 Les diagrammes de Voronoï revisités

Il reste à expliquer comment reconstituer le diagramme de Voronoï à partir de la triangulation de Delaunay.

C'est chose facile (voir la figure 11.7 à nouveau) : on considère tour à tour chaque arête $[pq]$ de la triangulation.

Alors, de deux choses l'une :

si l'arête est commune à deux triangles T_1 et T_2, on détermine les centres c_1 et c_2 de leurs cercles circonscrits, et on ajoute au diagramme de Voronoï l'arête $[c_1c_2]$ et les sommets c_1 et c_2 ;

si l'arête est sur la frontière de l'enveloppe convexe, elle n'appartient qu'à un triangle $[pqr]$; on détermine le centre c de son cercle circonscrit, et on ajoute au diagramme de Voronoï le sommet c et la demi-droite portée par la médiatrice de $[pq]$, d'extrémité c, qui se dirige vers l'extérieur de l'enveloppe convexe.

11.5 Implémentation en Caml

11.5.1 Quelques utilitaires

Programme 11.5 Quelques utilitaires pour l'algorithme de Tsai

```
let epsilon = 1.0e-8;;

let carré x = x *. x;;

let d2 (ax,ay) (bx,by) = carré(ax -. bx) +. carré(ay -. by);;

let cercle_circonscrit (ax,ay) (bx,by) (cx,cy) =
    let a = d2 (ax,ay) (0.,0.)
    and b = d2 (bx,by) (0.,0.)
    and c = d2 (cx,cy) (0.,0.)
    and q = 2. *. (ax *. (by-.cy) +. bx *. (cy-.ay) +. cx *. (ay-.by))
    in
    (   (* abscisse du centre *)
        (a *. (by-.cy) +. b *. (cy-.ay) +. c *. (ay-.by))/.q
    ,   (* ordonnée du centre *)
        -.(a *. (bx-.cx) +. b *. (cx-.ax) +. c *. (ax-.bx))/.q
    )
    ,   (* rayon du cercle *)
    sqrt((d2 (ax,ay) (bx,by))*.(d2 (bx,by) (cx,cy))*.(d2 (cx,cy) (ax,ay)))
    /. abs_float(q);;

let déterminant (ax,ay) (bx,by) = ax*.by -. ay*.bx;;

let vecteur (ax,ay) (bx,by) = (bx-.ax,by-.ay);;

let sens_direct a b c = 0. <. (déterminant (vecteur a b) (vecteur a c));;

let alignés a b c =
    abs_float(déterminant (vecteur a b) (vecteur a c)) <. epsilon;;

let est_triangle a b c = not (alignés a b c);;
```

Nous commençons par écrire quelques utilitaires d'usage général, qui font l'objet du fichier `tsai.util.ml` dont on fournit le listing : c'est le programme 11.5.

On trouve les fonctions `carré` et `d2` qui n'appellent pas de commentaires particuliers, pas plus que `déterminant`, `vecteur`, `sens_direct` ou `est_triangle`. Signalons simplement

qu'à cause des problèmes de précision habituels aux flottants, on a introduit une constante `epsilon` pour éviter dans `alignés` la comparaison avec `0.` qui est toujours hasardeuse.

un grand merci à Maple qui a donné les formules utilisées

La fonction `cercle_circonscrit` attend trois points représentés par des couples de coordonnées flottantes, et renvoie un couple formé du centre et du rayon du cercle circonscrit au triangle.

Programme 11.6 Algorithme de Tsai : 1ère étape

```
1  load_object "Cycles";;
2  #open "Cycles";;
3
4  load_object "Tri_fusion";;
5  #open "Tri_fusion";;
6
7  load_object "Convexe.float";;
8  #open "Convexe.float";;
9
10 include "tsai.util";;
11
12 let liste_vide = function [] -> true | _ -> false;;
13
14 let rec premier_d'accord prédicat = function
15       [] -> raise Not_found
16     | x :: q -> if prédicat(x) then x else premier_d'accord prédicat q;;
17
18 let rec discrimine prédicat = function
19       [] -> [],[]
20     | x :: q -> let oui,non = discrimine prédicat q
21                 in
22                 if prédicat(x) then (x::oui),non else oui,(x::non);;
23
24 let phase1 les_points =
25     let convexe = enveloppeConvexe les_points
26     in
27     let sommets_du_polygone = cycle_en_liste convexe
28     and arètes_du_polygone =
29         let rec crée_liste_des_arètes = function
30               a :: b :: q -> (a,b) :: crée_liste_des_arètes (b :: q)
31             | _ -> []
32         in crée_liste_des_arètes
33             ((valeur_cycle convexe) :: (cycle_en_liste (avance_cycle convexe)))
34     in
35     let sommets_internes = subtract les_points sommets_du_polygone
36     in
37     sommets_du_polygone , arètes_du_polygone , sommets_internes;;
```

On peut alors écrire la première partie de notre algorithme, ce qui constitue le programme 11.6.

il a fallu réécrire le programme de recherche de l'enveloppe convexe pour l'adapter au cas de points à coordonnées flottantes

On trouvera les incantations habituelles nécessaires pour pouvoir accéder à la structure de listes circulaires, au tri-fusion, et à la recherche de l'enveloppe convexe. On inclut également les utilitaires que nous venons de décrire.

On ajoute encore quelques fonctions d'intérêt général : `est_vide` dit si la liste qu'on lui passe en argument est vide ou non ; `premier_d'accord` prend en argument un prédicat et une liste, et renvoie le premier élément de la liste qui satisfait le prédicat ou déclenche l'exception `Not_found` s'il ne s'en trouve pas ; de façon analogue `discrimine`, avec les mêmes arguments, renvoie le couple des listes `oui` et `non` formées par les éléments qui satisfont ou non le prédicat.

un prédicat est du type `'a -> bool`

On est en position d'écrire `phase1` qui prend la liste des points du nuage et renvoie un triplet formé de la liste des sommets du polygone frontière de l'enveloppe convexe, de la liste des arêtes de ce polygone (sous la forme de couples de points), et de la liste des points du nuage qui sont à l'intérieur du polygone.

11.5.2 Deuxième étape de l'algorithme

Le programme 11.7 permet d'implémenter la deuxième phase de l'algorithme de Tsai. Il commence par définir quelques fonctions auxiliaires.

les utilitaires, c'est comme les lemmes, plus il y en a, et meilleur c'est . . . hummm

`test_de_Delaunay` prend en arguments trois points `p q r` et la liste de tous les sommets, et dit si disque (de frontière le) cercle circonscrit au triangle `pqr` ne contient bien aucun autre sommet que ces trois là. Notons que `mem` et `forall` sont des fonctions de la bibliothèque standard Caml.

la bibliothèque standard nous a déjà donné `subtract`

`nouveau_triangle p q r liste` vérifie que le triangle `pqr` n'est pas déjà dans la liste fournie, et que c'est bel et bien un triangle (*id est* pas trois points alignés).

On trouve ensuite `même_arête` qui est l'égalité entre arêtes (il n'y a pas d'ordre sur les extrémités qui définissent une arête) ; et `membre_arête` qui, à l'aide de la précédente, découvre si l'arête passée en premier argument fait partie de la liste d'arêtes passée en deuxième argument.

les mathématiciens notent $\setminus$ et Δ

`différence` et `différence_symétrique` effectuent les fonctions classiques sur les ensembles sur des listes d'arêtes. C'est le cœur de la deuxième phase de l'algorithme de Tsai : on ajoute les nouvelles arêtes si elles sont absentes, on les supprime si elles sont présentes, ce qui n'est qu'une formulation de la différence symétrique.

Il n'y a plus qu'à écrire `phase2` qui à partir de la liste des sommets du polygone convexe et de la liste de ses arêtes construit et renvoie la liste des triangles de sa triangulation de Delaunay. Elle utilise une procédure récursive `itère` qui épuise petit à petit son premier argument (la liste courante des arêtes à traiter) en ajoutant au fur et à mesure à son deuxième argument les triangles construits.

Programme 11.7 Algorithme de Tsai : 2e étape

```
38 let test_de_Delaunay p q r tous_les_sommets =
39     let centre,rayon = cercle_circonscrit p q r
40     in
41     let est_ok m = (mem m [p;q;r]) or (d2 centre m >. carré(rayon))
42     in
43     for_all est_ok tous_les_sommets ;;
44
45 let nouveau_triangle p q r anciens_triangles =
46     let autre_triangle (a,b,c) =
47         not (liste_vide (subtract [p;q;r] [a;b;c]))
48     in
49     (est_triangle p q r) &
50         (for_all autre_triangle anciens_triangles) ;;
51
52 let même_arête (a,b) (a',b') =
53     ( (a = a') & (b = b')) or ((a = b') & (b = a')) ;;
54
55 let rec membre_arête a = function
56       [] -> false
57     | t :: q -> (même_arête t a) or (membre_arête a q) ;;
58
59 let rec différence l1 l2 = match l1 with
60       [] -> []
61     | a :: q -> if membre_arête a l2 then différence q l2
62                 else a :: (différence q l2) ;;
63
64 let différence_symétrique l1 l2 =
65     let l = différence l1 l2
66     and m = différence l2 l1
67     in l @ m ;;
68
69 let phase2 les_sommets les_arêtes =
70     let rec itère arêtes triangles = match arêtes with
71           [] -> triangles
72         | (a,b) :: q
73             ->  let candidat x =    (nouveau_triangle a b x triangles)
74                                     & (test_de_Delaunay a b x les_sommets)
75                 in
76                 let c = premier_d'accord candidat les_sommets
77                 in
78                 itère   (différence_symétrique q [(b,c);(c,a)])
79                         ((a,b,c) :: triangles)
80     in
81     itère les_arêtes [] ;;
```

11.5.3 Troisième étape

On commence, dans le programme 11.8 qui implémente la troisième et dernière phase de l'algorithme de Tsai, encore une fois par quelques fonctions auxiliaires.

`chapeaute m (p,q,r)` dit si le triangle `pqr` chapeaute le point `m`, au sens où nous l'avons défini plus haut.

`supprime_doublons` renvoie la liste d'arêtes passée en argument privée de ceux de ses éléments qui y figureraient en plus d'un exemplaire. Pour ce faire elle commence par former la liste de ces éléments particuliers grâce à la fonction récursive `doublons`.

`liste_arêtes_de_triangles` prend une liste de triangles et renvoie la liste complète de toutes leurs arêtes.

`phase3` prend en arguments les listes des points déjà traités (au départ : les sommets de l'enveloppe convexe), des autres points du nuage, des triangles de la triangulation en cours (au départ : la triangulation du polygone obtenue grâce à `phase2`). Elle renvoie la liste des triangles de la triangulation de Delaunay du nuage.

Programme 11.8 Algorithme de Tsai : 3e étape

```
82 let chapeaute m (p,q,r) =
83     let centre,rayon = cercle_circonscrit p q r
84     in
85     (d2 m centre) <. carré(rayon);;
86
87 let supprime_doublons l =
88     let rec doublons = function
89           [] -> []
90         | a :: q -> if membre_arête a q then a :: (doublons q)
91                     else doublons q
92     in
93     différence l (doublons l);;
94
95 let rec liste_arêtes_de_triangles = function
96       [] -> []
97     | (a,b,c) :: q -> (a,b) :: (b,c) :: (c,a) ::
98                                   (liste_arêtes_de_triangles q);;
99
100 let rec phase3 pts_traités pts_restants triangles =
101     let triangles_après m =
102         let chapeautants,inertes = discrimine (chapeaute m) triangles
103         in
104         let arêtes = supprime_doublons
105                         (liste_arêtes_de_triangles chapeautants)
106         in
107         (map (function (a,b) -> (a,b,m)) arêtes) @ inertes
108     in
109     match pts_restants with
110           [] -> triangles
111         | m :: q -> phase3 (m :: pts_traités) q (triangles_après m);;
112
113 let tsai les_points =
114     let pts_polygone,arêtes_polygone,pts_internes = phase1 les_points
115     in
116     let triangles_polygone = phase2 pts_polygone arêtes_polygone
117     in
118     phase3 pts_polygone pts_internes triangles_polygone;;
```

Pour ce faire, elle a été écrite de façon récursive, l'arrêt se produisant quand il n'y a plus de points à traiter (ligne 110). Dans le cas général, on considère le premier point `m` à traiter, on construit la nouvelle liste des triangles et on itère (ligne 111). Bien sûr c'est `triangles_après` qui se charge de construire la nouvelle triangulation. Elle est définie en lignes 101–107. On discrimine tout d'abord ceux des anciens triangles qui chapeautent ou non `m` (ligne 102). On

supprime toutes les arêtes au moins en double de la liste de toutes les arêtes de ces triangles de la zone d'influence de m (lignes 104–105). Il n'y a plus qu'à ajouter aux triangles hors zone d'influence (leur liste est appelée `inertes`) les triangles obtenus à l'aide de m et des arêtes restantes, ce qui est fait en ligne 107.

La procédure finale `tsai` se contente d'enchaîner les trois phases de l'algorithme.

11.5.4 Détermination du diagramme de Voronoï

relis ce que nous avons dit en 11.4.6

On en arrive à la création du diagramme de Voronoï à partir de la triangulation de Delaunay.

Programme 11.9 Construction du diagramme de Voronoï

```
119 type voronoi = Segment of point * point | Demi_droite of point * direction
120 and direction == float * float ;;
121
122 let direction (ax,ay) (bx,by) c =
123     if sens_direct (ax,ay) (bx,by) c    then (by-.ay , ax-.bx)
124                                         else (ay-.by , bx-.ax) ;;
125
126 let rec présent (a,b) = function
127       [] -> raise Not_found
128     | ((p,q),_,c) :: l ->   if même_arête (a,b) (p,q) then c
129                             else présent (a,b) l ;;
130
131 let rec supprime (a,b) = function
132       [] -> []
133     | ((p,q),r,s) :: l ->   if même_arête (p,q) (a,b) then supprime (a,b) l
134                             else ((p,q),r,s) :: (supprime (a,b) l) ;;
135
136
137 let voronoi_et_delaunay les_points =
138     let rec combine = function
139           [] -> []
140         | ((a,b),c,centre) :: q
141             ->  try let centre' = présent (a,b) q
142                     in
143                     Segment(centre,centre') :: (combine (supprime (a,b) q))
144                 with
145                     Not_found ->
146                         Demi_droite(centre,(direction a b c)) :: (combine q)
147     and développe = function
148           [] -> []
149         | (a,b,c) :: q ->   let centre,_ = cercle_circonscrit a b c
150                             in
151                             ((a,b),c,centre) :: ((b,c),a,centre) ::
152                             ((c,a),b,centre) :: (développe q)
153     in
154     let delaunay = tsai les_points
155     in
156     let les_arêtes = développe delaunay
157     in
158     let diagramme = combine les_arêtes
159     in
160     delaunay , diagramme ;;
```

Les éléments du diagramme de Voronoï sont de deux types, les segments, caractérisés par les deux points extrémités, et les demi-droites, caractérisées par leur unique point extrémité, et un vecteur donnant la direction de la demi-droite ; c'est ce que traduisent les types Caml définis en lignes 119–120.

Pour les demi-droites, nous aurons besoin de savoir déterminer la direction de la médiatrice de $[ab]$ qui "sort du triangle" (abc). Tout dépend en fait de l'orientation du repère $(a, \overrightarrow{ab}, \overrightarrow{ac})$, c'est ce que réalise la procédure `direction`.

enfin, tous. . . ?

Avant de commenter `présent` et `supprime`, intéressons-nous à la fonction objet de tous nos désirs : `voronoi_et_delaunay` qui prend en argument la liste des points du nuage et renvoie un couple formé de la liste des triangles de la triangulation de Delaunay et de la liste des objets constitutifs du diagramme de Voronoï (segments et demi-droites).

Après avoir appelé `tsai` en ligne 154, on applique à la liste des triangles obtenus la fonction `développe` (définie en lignes 147–152) qui se charge de créer une liste d'éléments de la forme `((a,b),c,centre)` pour chaque arête $[ab]$ de chaque triangle. Bien sûr `centre` représente le centre du cercle circonscrit associé au triangle, qu'on ne calcule ainsi qu'une fois pour chaque. Mais pourquoi conserver aussi le troisième point c du triangle? tout simplement pour pouvoir orienter (comme il a été dit plus haut) la demi-droite correspondante dans le cas où $[ab]$ est un côté du polygone frontière de l'enveloppe convexe du nuage.

bonne question, merci de me l'avoir posée !

Que fait-on ensuite? on applique mot à mot ce qui a été expliqué en 11.4.6, en considérant tour à tour les arêtes ainsi listées.

Si une arête figure deux fois, c'est à dire si elle est présente dans la liste des arêtes non encore considérées, elle figure en association avec un nouveau centre : il suffit de construire le segment qui relie ces deux centres, et d'appliquer récursivement `combine` à la liste des arêtes débarrassée de ces deux arêtes maintenant traitées. C'est ce qu'on fait en lignes 140–143. On voit maintenant ce que fait `présent (a,b) liste` : si ab figure dans la liste des arêtes fournie on renvoie le centre de cercle circonscrit associé, et sinon on déclenche l'exception `Not_found`. Quant à `supprime`, elle enlève d'une liste d'arête tout triplet `((c,d),e,centre')` tel que (c, d) et (a, b) soient la même arête.

Sinon, on construit la demi-droite voulue avant de réaliser l'appel récursif (lignes 145–146).

11.5.5 Affichage dans une fenêtre graphique

Nous donnons le petit programme 11.10 qui permet, en usant de la bibliothèque graphique de Caml, d'afficher diagramme de Voronoï et triangulation de Delaunay d'un nuage de points aléatoire.

Nous commençons par redéfinir les fonctions de tracé de base pour qu'elles s'appliquent aux points à coordonnées flottantes. En outre, en les combinant, on écrit petit à petit les fonctions de tracé de cercles circonscrits, ou d'un polygone, etc. Notons que nous avons choisi de dessiner les triangulations en trait gras et les diagrammes en trait fin. De plus chaque point du nuage est représenté par un petit disque de rayon 3 pixels.

On écrit `liste_aléatoire` qui renvoie une liste aléatoires de points en vérifiant qu'ils tiennent dans la fenêtre graphique de Caml et qu'elle n'a pas deux points égaux.

La procédure `affiche` réalise alors l'affichage de figures pour un nombre fixé de nuages aléatoires de taille également fixée. Il n'y a plus qu'à se faire plaisir, en regardant l'écran. . .

On trouvera dans la figure 11.11 une copie de l'écran de mon Macintosh lors d'une exécution de ce programme.

Programme 11.10 Affichage graphique des diagrammes de Voronoï de nuages de points aléatoires

```
include "Tsai.ml";;
#open "graphics";;

random__init 0;;
open_graph "";; (* 480 * 280 *)

let moveTo (ax,ay) = moveto (int_of_float ax) (int_of_float ay);;
let lineTo (ax,ay) = lineto (int_of_float ax) (int_of_float ay);;
let trace_triangle a b c = moveTo a; lineTo b; lineTo c; lineTo a;;
let trace_cercle (ax,ay) r =
    draw_circle (int_of_float ax) (int_of_float ay) (int_of_float r);;
let trace_cercle_circonscrit a b c =
    let o,r = cercle_circonscrit a b c in trace_cercle o r;;
let trace_point (ax,ay) = fill_circle (int_of_float ax)  (int_of_float ay) 3;;
let trace_liste_de_points l = do_list trace_point l;;
let traceTriangle (a,b,c) = trace_liste_de_points [ a; b; c ];
                            trace_triangle a b c;;
let infini (ax,ay) (dx,dy) = (ax +. 20.*.dx,ay +. 20.*.dy);;
let trace_objet_du_diagramme = function
      Segment(a,b) -> moveTo a; lineTo b
    | Demi_droite(a,d) -> moveTo a; lineTo (infini a d);;

exception Gagné of point list;;

let rec liste_aléatoire = function
      0 -> []
    | taille -> let l = liste_aléatoire (taille - 1)
                in
                try while true do
                    let a = (120.+.(random__float 240.),20.+.(random__float 240.))
                    in
                    if not (mem a l) then raise (Gagné(a :: l))
                    done;
                    l
                with Gagné res -> res;;

let attend () = wait_next_event [ Button_down; Key_pressed ]; ();;

let affiche nb_points nb_images =
    for i = 1 to nb_images do
        clear_graph ();
        let l = liste_aléatoire nb_points
        in
        let t,v = voronoi_et_delaunay l
        in
        set_line_width 2; map traceTriangle t;
        set_line_width 1; map trace_objet_du_diagramme v;
        if i < nb_images then attend ()
    done;;
```

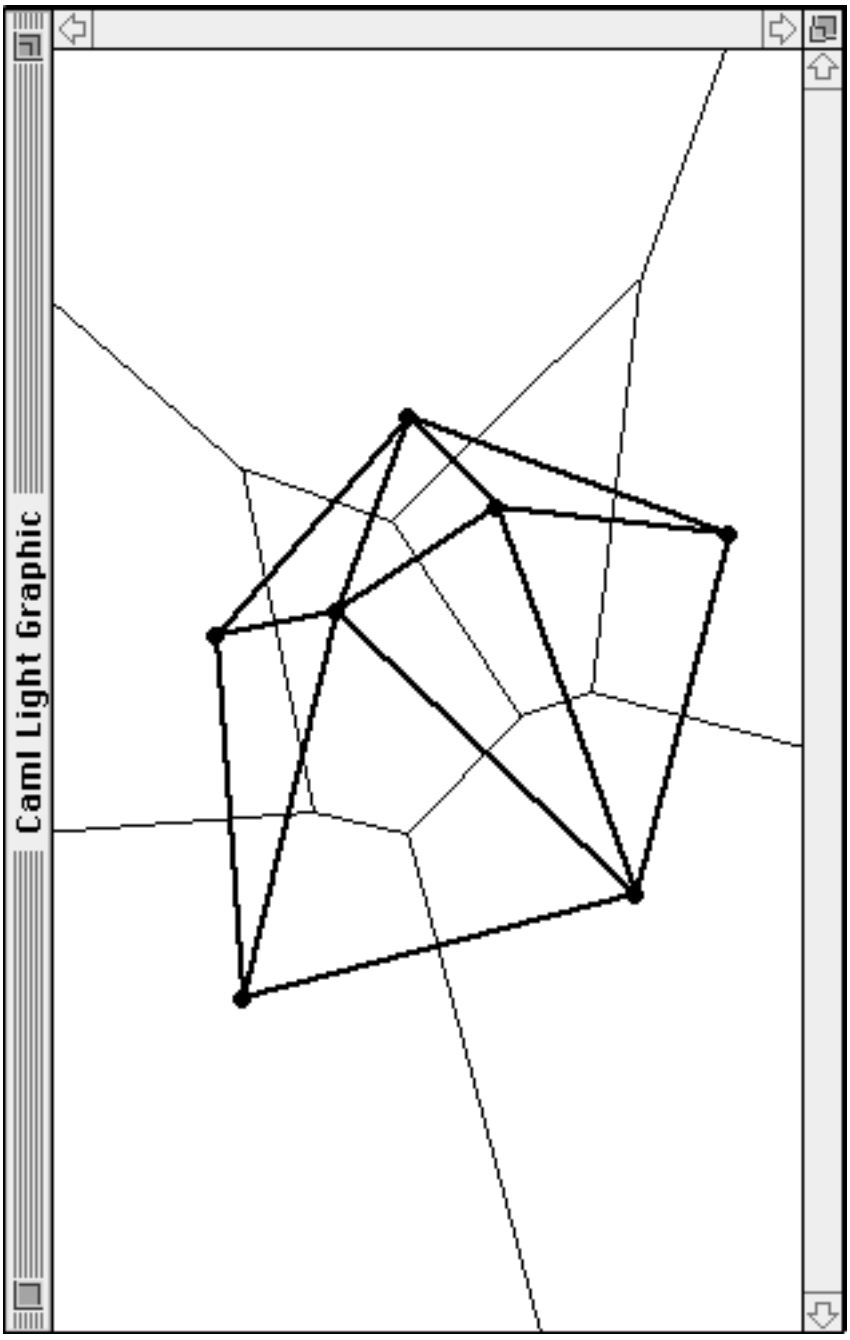


Figure 11.11: Une copie d'écran de mon Macintosh

Partie IV

Annexes

Annexe A

Un petit manuel de référence de Caml

A.1 Types de base

A.1.1 Unit

Le type `unit` ne contient qu'un élément, noté `()`. On l'utilise pour les fonctions "sans arguments" (en fait, elles en ont un, c'est `()`). Exemple : `quit : unit -> unit` qui s'invoque par `quit() ;;` On l'utilise aussi pour les fonctions qui ne renvoient "rien" (en fait, elles renvoient quelque chose : `()`). Exemple : les fonctions d'impression.

A.1.2 Booléens

Le type `bool` contient deux valeurs : `true` (vrai) et `false` (faux). Les opérateurs sur les booléens sont : `&` et `or` qui seront détaillés en A.3.2 (ce sont, bien sûr, le ET et le OU), `not` (la négation). Tous ces opérateurs sont de faible priorité : on s'astreindra à parenthéser complètement les expressions booléennes.

depuis la version 0.7 de Caml, nous disposons d'une nouvelle orthographe : `&&` et `||`

A.1.3 Nombres entiers

Le type `int` correspond aux entiers (signés). Les opérateurs correspondants sont : `+`, `-`, `*`, `/`, `mod`. La division est une division entière ; exemple : `7/3` s'évalue en `2` ; `7 mod 3` donne `1`, le reste de la division précédente.

`succ` et `pred` sont respectivement les fonctions successeur et prédécesseur.

On dispose des comparaisons standards : `< <= = > >= <>` et de la valeur absolue `abs` ainsi que de `min` et `max` qui attendent deux arguments.

A.1.4 Nombres à virgule ou "flottants"

Le type `float` correspond aux nombres "à virgule", on les reconnaît (paradoxalement) à la présence du point : `1.2e6` correspond à $1{,}2\,10^6$. Attention, `1` est un `int` alors que `1.` est un `float`.

On dispose des opérateurs courants : `+. -. *. /.` et des comparaisons : `<. <=. =. >. >=. <>.`

Les fonctions transcendantes s'orthographient : `sin cos tan asin acos atan sqrt log exp`. Le logarithme (`log`) est népérien et `sqrt` désigne la racine carrée *(square root)*. L'élévation à la puissance s'obtient par `power : float -> float -> float` ; la valeur absolue par `abs_float`.

Enfin, les conversions entiers/flottants sont

```
int_of_float : float -> int  et  float_of_int : int -> float
```

qui font ce qu'elles peuvent.

A.1.5 Caractères

Le type `char` correspond aux caractères. On les écrit `'a' 'b'` ou encore `';'` par exemple. Les caractères spéciaux courants sont : `'\\'` et `'\''` et `'\r'` et `'\t'` pour, respectivement, le backslash, le backquote (ou accent grave), le retour-chariot et la tabulation.

Les seuls opérateurs de comparaison disponibles sont = et <>

Pour les conversions on dispose de

```
int_of_char : char -> int  et  char_of_int : int -> char
```

qui s'appuient sur le codage ascii des caractères.

A.1.6 Chaînes de caractères

Le type `string` correspond aux chaînes de caractères. La chaîne `abc"def` est notée `"abc\"def"` ; les caractères spéciaux vus précédemment sont autorisés.

Les opérateurs correspondants sont `^` pour la concaténation et `=` et `<>` pour les comparaisons.

La bibliothèque standard fournit entre autres les fonctions suivantes.

- `string_length : string -> int` renvoie la longueur d'une chaîne ;
- `nth_char : string -> int -> char` renvoie le n-ième caractère d'une chaîne (la numérotation commence à zéro) ;
- `set_nth_char : string -> int -> char -> unit` modifie le n-ième caractère d'une chaîne ;
- `sub_string : string -> int -> int -> string` renvoie une nouvelle chaîne : par exemple, `sub_string "abcdef" 2 3` donne `"cde"` qui commence au 2^{e} caractère et est de longueur 3 ;
- `make_string : int -> char -> string` crée une chaîne dont on précise la taille et le caractère de remplissage.

c'est bien plus pratique

Depuis la version 0.7, on dispose d'une nouvelle syntaxe : si `s` est une chaîne, si `i` est un indice dans la chaîne, `s.[i]` désigne le caractère correspondant de la chaîne, et `s.[i] <- c` permet de le modifier.

Les fonctions de comparaison sont toutes du type `string -> string -> bool` ; elles s'appuient sur l'ordre lexicographique et se notent respectivement : `eq_string` (égalité), `neq_string` (inégalité), `ge_string` (relation supérieur ou égal), `le_string` (relation inférieur ou égal), `gt_string` (relation supérieur strict), `lt_string` (relation inférieur strict).

A.1.7 Couples et multiplets

Les couples correspondent au produit cartésien de deux types. Exemple : `(1,"asd")` est du type `int * string`. Bien sûr, on dispose de même des multiplets comme `(false, 3, "asd")` du type `bool * int * string`.

La bibliothèque standard fournit entre autres les fonctions suivantes.

- `fst : 'a * 'b -> 'a` renvoie le premier élément d'un couple ;
- `snd : 'a * 'b -> 'b` renvoie le second ;
- `split : ('a * 'b) list -> 'a list * 'b list` prend une liste de couples et renvoie le couple des listes formées des premiers et des seconds éléments ;
- `combine : 'a list * 'b list -> ('a * 'b) list` effectue l'opération inverse ;
- `map_combine : ('a * 'b -> 'c) -> 'a list * 'b list -> 'c list` fait ce que l'on pense :

```
#map_combine (function (x,y) -> x*y) ([1; 2; 3], [4; 5; 6]) ;;
- : int list = [4; 10; 18]
```

- `do_list_combine : ('a * 'b -> 'c) -> 'a list * 'b list -> unit` fait de même, mais ne renvoie rien : n'a d'intérêt que si la fonction passée en premier argument réalise des effets de bord.

A.1.8 Listes

Une liste se note `[x1; x2; x3; ...; xN]` où les objets `xi` sont tous d'un même type `'a`, le type de la liste est alors `'a list`. La liste vide se note `[]` ; l'opérateur de construction des liste est `::` qu'on lit *quatre points*, ainsi `1 :: 2 :: 3 :: []` ou `1 :: 2 :: [3]` ou `1 :: [2; 3]` valent `[1; 2; 3]`, du type `int list`. On dit que l'opérateur `::` est associatif à droite. Les deux fonctions d'accès fondamentales sont `hd` (pour h*ea*d) qui renvoie la tête de la liste, et `tl` (pour t*ai*l) qui renvoie la queue de la liste. Ces deux fonctions déclenchent `Failure "hd"` ou `Failure "tl"` si on tente de les appliquer à la liste vide. La plupart du temps, on leur préfèrera le filtrage.

L'opérateur de concaténation des listes se note `@` ; son exécution ne se fait pas en temps constant mais linéaire en la taille de la première liste.

La bibliothèque standard fournit entre autres les fonctions suivantes.

- `list_length : 'a list -> int` renvoie la longueur de la liste ;
- `rev : 'a list -> 'a list` fabrique une copie renversée de la liste passée en argument, ainsi `rev [1; 2; 3]` vaut `rev [3; 2; 1]` ;
- `map : ('a -> 'b) -> 'a list -> 'b list` construit la liste des résultats de l'application de la fonction donnée en premier argument à chacun des éléments de la liste passée en second argument ;
- `do_list : ('a -> 'b) -> 'a list -> unit` fait comme `map` mais sans renvoyer aucun résultat : n'est intéressante que pour les effets de bord ;
- `for_all : ('a -> bool) -> 'a list -> bool` dit si le prédicat (fonction à valeur booléenne) passé en premier argument est satisfait par tous les éléments de la liste passée en second argument ;
- `exists : ('a -> bool) -> 'a list -> bool` dit si le prédicat (fonction à valeur booléenne) passé en premier argument est satisfait par au moins un élément de la liste passée en second argument ;
- `mem : 'a -> 'a list -> bool` dit si l'élément passé en premier argument est élément de la liste passée en second argument ;
- `except : 'a -> 'a list -> 'a list` renvoie la liste passée en deuxième argument privée du premier de ses éléments égal à l'objet passé en premier argument ;
- `substract : 'a list -> 'a list -> 'a list` renvoie la liste des éléments de la première liste qui ne figurent pas dans la seconde ;
- `union : 'a list -> 'a list -> 'a list` renvoie la deuxième liste augmentée des éléments de la première qui n'y figurent pas déjà ;
- `intersect : 'a list -> 'a list -> 'a list` renvoie une copie de la première liste privée des éléments qui ne figurent pas dans la seconde ;
- `index : 'a -> 'a list -> int` renvoie la position de la première occurrence du premier argument dans la liste passée en second argument, la numérotation commençant à 0 ;
- `assoc : 'a -> ('a * 'b) list -> 'b` gère les listes associatives : par exemple `assoc 2 [(1,'a'); (2,'b') ; (2,'c') ; (3,'d')]` vaut `'b'`. Déclenche l'exception `Not_found` en cas d'échec.

A.2 Types construits

A.2.1 Types produit

Comme les multiplets, les types produit sont des produits cartésiens, mais ils permettent en outre de nommer les projections, qu'on appelle des *champs*.

Par exemple

```
type individu = { Nom : string ; Prénom : string ; Sécu : int }
```

définit un type produit qui comporte trois champs. L'accès aux différents champs (c'est-à-dire les différentes projections) répond à la syntaxe suivante : si `moi` est un objet de type `individu`, on accède à ses projections par `moi.Nom`, `moi.Prénom`, `moi.Sécu`. Un objet de ce type est par exemple `{ Nom = "Lanturlu" ; Prénom = "Anselme" ; Sécu = 123042 }`. On notera que le filtrage ne nécessite pas l'explicitation de tous les champs, alors qu'elle est indispensable lors de la définition d'un objet.

A.2.2 Types somme

Mathématiquement parlant, il s'agit ici de sommes disjointes.

Par exemple `type booléen = Vrai | Faux` est une définition parfaitement satisfaisante des booléens, `Vrai` et `Faux` sont des *constructeurs* du type. Attention : la définition suivante n'est pas licite `type nombre = int | float` ; il faut passer par l'usage des constructeurs, en écrivant par exemple `type nombre = Entier of int | Flottant of float`. Un objet de ce type se note `Entier(123)` ou `Flottant(3.14159)`. Le filtrage permet de discriminer les valeurs.

A.2.3 Types paramétrés

Toute définition de type construit peut être paramétrée par des variables de types.

Par exemple `type 'a numéroté = int * 'a`, ou, pour définir un type analogue aux couples, `type ('a,'b) paire = {Premier : 'a ; Second : 'b}`. Ou encore, pour les types somme, par exemple

```
type ('a,'b) simple_ou_double = Simple of 'a | Double of 'a * 'b
```

qui permet de décrire quelque chose comme l'union disjointe de A et $A \times B$.

A.2.4 Types (mutuellement) récursifs

Donnons quelques exemples :

```
type 'a liste = Nil | Cons of 'a * 'a liste
```

permettrait de définir un type paramétré semblable au type prédéfini Caml `'a list`.

```
type 'a arbre = Nil | Nœud of 'a nœud_interne
 and 'a nœud_interne = { Clé : 'a ; Fils_droit : 'a arbre ; Fils_gauche : 'a arbre }
```

fournit une description d'un arbre binaire.

A.2.5 Abréviations de type

Caml nous permet de définir des synonymes pour des types simples.

Par exemple `type point3d == float * float * float` permet d'invoquer par la suite cette abréviation pour définir un nouveau type, ou pour une coercion de type. Plus simplement, cela offre une interprétation lisible d'un type complexe.

A.3 Structures de contrôle

A.3.1 Séquencement

Une séquence d'expressions est constituée d'une suite d'expressions séparées par des points-virgules. Sa valeur est celle de la dernière expression évaluée. Cela n'a d'intérêt que si les premières expressions réalisent des effets de bord. Le plus souvent, on encadre une séquence par le couple `begin-end`, ou, à la rigueur, par une paire de parenthèses.

A.3.2 Opérateurs `&` et `or`

Les opérateurs booléens `&` et `or` ne peuvent pas être définis en tant que fonctions. En effet, ils n'évaluent leurs arguments qu'au fur et à mesure de leur besoin. Par exemple, `true or (raise (Failure "planté !"))` ne déclenche pas d'erreur mais vaut `true` ; de même `false & ((3/0) > 0)` ne déclenche pas d'erreur mais vaut `false`.

Notons qu'à partir de la version 0.7 de Caml, nous disposons d'une alternative pour l'orthographe de ces opérateurs, à savoir `&&` et `||`. *un emprunt au langage C*

A.3.3 Conditionnelles

Une expression conditionnelle est de la forme

`if` *expression_test* `then` *expression_1* `else` *expression_2*

où *expression_test* est de type `bool` et où *expression_1* et *expression_2* doivent être d'un même type, qui sera celui de l'expression conditionnelle toute entière.

Dans le seul cas où le type de *expression_1* est `unit` la clause `else` peut être omise.

A.3.4 Filtrage

L'expression `match`

Sa syntaxe est :

`match` *expression* `with`
motif1 `->` *expr1*
`|` *motif2* `->` *expr2*
...
`|` *motifN* `->` *exprN*

Caml tente de filtrer *expression* avec *motif1* ; en cas de succès, l'expression `match` complète vaut *expression1*. En cas d'échec, les motifs suivants sont essayés dans l'ordre. Si aucun motif ne filtre *expression*, l'exception `Match_failure` est déclenchée. Il faut que toutes les expressions *expri* soient du même type, qui est celui de l'expression `match` toute entière.

Par exemple, le filtrage typique sur les listes est :

```
match liste with
     [] -> ...
   | tête :: queue -> ...
```

Motifs de filtrage

Les motifs sont définis par :

motif ≡
  _
  | *identificateur*
  | *constante*
  | `[]`
  | `[` *motif* ; . . . ; *motif* `]`
  | *motif* :: *motif*
  | *motif*, . . . , *motif*
  | `{` *étiquette* = *motif* ; . . . ; *étiquette* = *motif* `}`
  | *constructeur-constant*
  | *constructeur-non-constant motif*
  | (*motif*)
  | *motif* | *motif*
  | *motif* as *identificateur*
  | (*motif* : *expression-de-type*)

- _ filtre tout et n'importe quoi ;
- un identificateur filtre tout mais en effectuant la liaison ;
- une constante ne filtre qu'elle-même ;
- `[]` filtre la liste vide ;
- une liste de motifs filtre, motif par motif, les éléments d'une liste ;
- *motif1* :: *motif2* filtre une liste dont la tête est filtrée par *motif1* et la queue par *motif2* ;
- un multiplet de motifs filtre, motif par motif, les éléments d'un multiplet ;
- `{` *étiquette* = *motif* ; ... ; *étiquette* = *motif* `}` filtre champ par champ une expression de type produit ;
- un constructeur constant ne filtre que lui-même ;
- *constructeur-non-constant motif* filtre une expression de type somme ;
- on peut parenthéser les motifs ;
- *motif1* | *motif2* filtre ce que filtrent *motif1* ou *motif2*, qui n'ont pas le droit de procéder à des liaisons ;
- *motif* as *identificateur* filtre ce que filtre *motif* et introduit la liaison de l'expression filtrée avec *identificateur* ;
- (*motif* : *expression-de-type*) filtre une expression filtrée par *motif* qui a le type *expression-de-type*.

guards en anglais

Depuis la version 0.7, Caml reconnaît ce qu'on peut appeler des *surveillants de motifs* qui peuvent s'ajouter après chaque motif. Avec une grammaire BNF, on écrirait

motif_avec_surveillants ≡
  motif
  | *motif* when *cond*

Dans la nouvelle syntaxe, le motif invoqué n'est réputé filtrer l'expression testée que si la condition écrite s'évalue en `true`, avec les liaisons éventuellement introduites par le filtrage, bien sûr. À défaut, on passe, comme d'habitude, au motif suivant.

Expressions fonctionnelles

Une fonction se définit naturellement par filtrage :

```
function   motif1 -> expr1
         | motif2 -> expr2
           ...
         | motifN -> exprN
```

Si `'a` est le type le plus général des expressions filtrées par les *motifi* et si `'b` est le type commun aux *expri*, le type de cet objet (fonctionnel) se note `'a -> 'b`.

Remarques :

- l'air de rien `function x -> x` fait déjà intervenir un filtrage ;
- l'expression

  ```
  match expression with
      motif1 -> expr1
    | motif2 -> expr2
      ...
    | motifN -> exprN
  ```

  est en fait équivalente à

  ```
  ( function   motif1 -> expr1
             | motif2 -> expr2
               ...
             | motifN -> exprN ) ( expression )
  ```

- `function` *motif1* `->` ... `->` `function` *motifN* `->` *expression*

  peut s'abréger en

  `fun` *motif1 ... motifN* `->` *expression*

- `->` est associatif à droite, c'est-à-dire que le type `'a -> 'b -> 'c` doit s'entendre comme étant le type `'a -> ( 'b -> 'c )`
- Caml comprend l'expression `x y z` comme étant `(x y) z` et non pas `x (y z)`.

Expressions `let` locales

L'expression

> `let` *motif1* = *expression1*
> and ...
> and *motifN* = *expressionN*
> in *expression*

opère des liaisons (par filtrage) qui ne seront que locales à l'évaluation de *expression*. On notera que toutes les *expressioni* sont évaluées avant la première liaison. Elle pourrait s'écrire

> `(` `fun` *motif1 ... motifN* `->` *expression* `)` *expression1 ... expressionN*

En écrivant `let rec` au lieu de `let`, on autorise des définitions mutuellement récursives.

Expressions `let` globales : définitions

L'expression

> `let` *motif1* = *expression1*
> and ...
> and *motifN* = *expressionN*

opère des liaisons (par filtrage) qui seront globales : on ajoute de nouvelles définitions à l'interprète. On notera que toutes les *expressioni* sont évaluées avant la première liaison. Ainsi :

```
#let x = 0 and y = 1 ;;
x : int = 0
y : int = 1
#let x = y and y = x ;;
x : int = 1
y : int = 0
```

En écrivant `let rec` au lieu de `let`, on autorise des définitions mutuellement récursives.

Sucre syntaxique

> `let f` *motif1 ... motifN* `=` *expression*

est compris par Caml comme

> `let f = fun` *motif1 ... motifN* `->` *expression*.

A.3.5 Boucles

On dispose de deux types de boucles. La boucle `while` obéit à la syntaxe suivante

> `while` *expression_test* `do` *expression* `done`

La boucle `for` obéit à la syntaxe suivante

> `for` *identificateur* `=` *expr1* `to` *expr2* `do` *expression* `done`

où *expr1* et *expr2* sont du type `int`. L'identificateur est implicitement local. Caml connaît aussi la version `downto`.

A.3.6 Exceptions

Il y a deux façons de définir des exceptions : `exception` *identificateur* et `exception` *identificateur* `of` *expression-de-type*. Les définitions d'exception ajoutent de nouveaux constructeurs au type somme prédéfini `exn` des valeurs "exceptionnelles".

On déclenche une exception en invoquant la fonction `raise : exn -> 'a`. On rattrape une exception par la structure de contrôle `try`, dont la syntaxe est :

`try` *expression* `with` *motif1* `->` *expression1*
`|`...
`|` *motifN* `->` *expressionN*

L'évaluation de *expression* peut déclencher une exception, celle-ci est éventuellement interceptée par le filtrage du `try` (sans quoi elle se propage), l'évaluation de *expression* se poursuit alors par l'évaluation du *expri* adéquat.

Il peut être intéressant d'utiliser des exceptions non constantes pour récupérer l'état du calcul au moment du déclenchement de l'exception.

Par exemple :

```
#exception Trouvé of int ;;
Exception Trouvé defined.
#let trouve_indice prédicat =
    let rec trouve n = function
          [] -> raise ( Failure "absent" )
        | a :: q -> if prédicat(a) then raise (Trouvé n)
                    else trouve (n+1) q
    in trouve 0 ;;
trouve_indice : ('a -> bool) -> 'a list -> 'b = <fun>
#trouve_indice (function x -> (x mod 2) = 0) [ 3 ; 5 ; 8 ; 7 ] ;;
Uncaught exception: Trouvé 2
```

L'exception `Failure of string` est prédéfinie. `failwith` *chaîne* est équivalent à `raise ( Failure` *chaîne* `)`.

A.4 Effets de bord : types mutables

A.4.1 Types mutables

Caml permet de définir des valeurs mutables de type produit en introduisant le mot-clef `mutable` devant un (ou plusieurs) champ d'un type produit ; on dispose alors de l'opération d'affectation, qui permet de modifier physiquement les valeurs correspondantes. Exemple :

```
#let toto = {Secu = 111 ; Age = 14} ;;
toto : personne = {Secu=111; Age=14}
#toto.Age ;;
- : int = 14
#toto.Age <- 15 ;;
- : unit = ()
#toto ;;
- : personne = {Secu=111; Age=15}
```

A.4.2 Autre type mutable : les références

Si `'a` est un type, le type `'a ref` est le type des références de ce type. On dispose des deux opérateurs `!` de déréférencement, et `:=` d'affectation. Voici une petite session Caml pour en montrer l'usage :

```
#let a = ref 1 ;;
a : int ref = ref 1

#a := ((!a) + 1) ;;
```

```
- : unit = ()

#a ;;
- : int ref = ref 2

#!a ;;
- : int = 2
```

Voici comment pourraient être définies les références :

```
#type 'a réference = { mutable Réference : 'a } ;;
Type réference defined.

#let référence x = { Réference = x } ;;
référence : 'a -> 'a réference = <fun>

#let déréf {Réference = x} = x ;;
déréf : 'a réference -> 'a = <fun>

#let reçoit r x = r.Réference <- x ;;
reçoit : 'a réference -> 'a -> unit = <fun>

##infix "reçoit" ;;

#let a = référence 1 ;;
a : int réference = {Réference=1}

#a reçoit ((déréf a) + 1) ;;
- : unit = ()

#a ;;
- : int réference = {Réference=2}

#déréf a ;;
- : int = 2
```

A.4.3 Autre type mutable : les vecteurs

Un vecteur se note `[| x1; x2; x3; ...; xN |]` où les objets `xi` sont tous d'un même type `'a`, le type du vecteur est alors `'a vect`. Si `v : 'a vect` est un vecteur, on accède à son `i`-ème élément par `v.(i)`, la numérotation commençant à 0 ; la structure est mutable : on modifie en place le `i`-ème élément en écrivant `v.(i) <-` *`expression`*.

La bibliothèque standard fournit entre autres les fonctions suivantes.

- `vect_length : 'a vect -> int` renvoie la taille du vecteur ;
- `make_vect : int -> 'a -> 'a vect` fabrique un nouveau vecteur dont la taille est fixée par le premier argument, le second initialisant chacun de ses éléments ;
- `list_of_vect : 'a vect -> 'a list` construit la liste des éléments du vecteur ;
- `vect_of_list : 'a list -> 'a vect` réalise l'opération inverse.

A.5 Effets de bord : entrées / sorties

A.5.1 Entrées/sorties clavier/écran

La bibliothèque standard fournit entre autres les fonctions suivantes d'affichage à l'écran :

- `print_char : char -> unit` affiche un caractère à l'écran ;
- `print_string : string -> unit` affiche une chaîne de caractères à l'écran ;
- `print_int : int -> unit` affiche la représentation décimale d'un entier à l'écran ;
- `print_float : float -> unit` affiche la représentation décimale d'un flottant à l'écran ;
- `print_newline : unit -> unit` affiche un saut de ligne à l'écran.

Signalons au passage l'existence de quatre fonctions de conversion :

`string_of_int : int -> string` et `string_of_float : float -> string`,

et leurs inverses

`int_of_string : string -> int` et `float_of_string : string -> float`.

La bibliothèque standard fournit entre autres les fonctions suivantes d'entrée au clavier :

- `read_line : unit -> string` attend l'entrée au clavier d'une chaîne de caractères limitée par un saut de ligne, renvoie cette chaîne sans le saut de ligne ;
- `read_int : unit -> int` est la composée de `int_of_string` sur `read_line` ;
- `read_float : unit -> float` est la composée de `float_of_string` sur `read_line` ;

A.5.2 Fichiers texte

Les fichiers texte utilisent le format ascii standard et sont traitables par tous les éditeurs de texte.

La bibliothèque standard fournit entre autres les fonctions d'entrée suivantes :

- `open_in : string -> in_channel` prend le nom du fichier à ouvrir et fournit un objet de type `in_channel` via lequel se feront toutes les opérations d'entrée sur ce fichier ;
- `input_char : in_channel -> char` lit un caractère sur le fichier ;
- `input_line : in_channel -> string` lit sur le fichier une chaîne de caractères limitée par un saut de ligne, renvoie cette chaîne sans le saut de ligne ;
- `close_in : in_channel -> unit` ferme le fichier en lecture.

Toute lecture qui tente d'aller au delà de la fin du fichier déclenche l'exception `End_of_file`.

La bibliothèque standard fournit entre autres les fonctions de sortie suivantes :

- `open_out : string -> out_channel` prend le nom du fichier à ouvrir et fournit un objet de type `in_channel` via lequel se feront toutes les opérations de sortie sur ce fichier ;
- `output_char : out_channel -> char -> unit` écrit un caractère sur le fichier ;
- `output_string : out_channel -> string -> unit` écrit une chaîne de caractères sur le fichier ;
- `close_out : out_channel -> unit` ferme le fichier en écriture.

A.5.3 Fichiers binaires

Les fichiers binaires utilisent un format particulier à Caml pour représenter les valeurs du langage, et ne sont lisibles que par des programmes Caml.

La bibliothèque standard fournit entre autres les fonctions d'entrée suivantes :

- `open_in_bin : string -> in_channel` prend le nom du fichier binaire à ouvrir et fournit un objet de type `in_channel` via lequel se feront toutes les opérations d'entrée sur ce fichier ;
- `input_value : in_channel -> 'a` lit une valeur Caml sur le fichier ;
- `close_in : in_channel -> unit` ferme le fichier en lecture.

L'exception `End_of_file` est déclenchée dans les mêmes conditions que précédemment.

La bibliothèque standard fournit entre autres les fonctions de sortie suivantes :

- `open_out_bin : string -> out_channel` prend le nom du fichier binaire à ouvrir et fournit un objet de type `in_channel` via lequel se feront toutes les opérations de sortie sur ce fichier ;
- `output_value : out_channel -> 'a -> unit` écrit une valeur Caml sur le fichier ;
- `close_out : out_channel -> unit` ferme le fichier en écriture.

A.6 Effets de bord : la bibliothèque graphique

La bibliothèque graphique n'est accessible qu'après l'incantation `#open "Graphics" ;;` On ouvre la fenêtre graphique par l'incantation `open_graph "highest"` qui renvoie `()`.

On dispose alors entre autres des fonctions et types suivants :

- le type `color` est prédéfini comme un synonyme de `int`. La fonction `rgb` attend trois entiers en arguments et renvoie la couleur décrite par ce codage RVB. `black white red green blue yellow cyan magenta` sont prédéfinies ;
- `set_color : color -> unit` fixe la couleur courante des tracés ;
- `clear_graph : unit -> unit` efface la fenêtre graphique ;
- `size_x : unit -> int` et `size_y : unit -> int` renvoient la taille de la fenêtre graphique ;
- `plot : int -> int -> unit` affiche un point ;
- `moveto : int -> int -> unit` déplace le point courant sans dessiner ;
- `lineto : int -> int -> unit` trace un segment entre le point courant et le point spécifié qui devient le nouveau point courant ;
- `current_point : unit -> int * int` renvoie la position du point courant ;
- `draw_circle : int -> int -> int -> unit` attend les coordonnées du centre et le rayon du cercle à tracer ;
- `fill_rect x y largeur hauteur` peint un rectangle ;
- `fill_circle x y rayon` peint un disque ;
- `wait_next_event [ Button_down ; Key_pressed ]` attend sans rien faire que l'utilisateur déclenche un événement clavier ou souris, peu importe la valeur renvoyée ;
- `close_graph : unit -> unit` supprime la fenêtre graphique.

Annexe B

Quelques idées pour la programmation fonctionnelle

B.1 Fonctionnelles en guise de structures de contrôle

Nous allons ici montrer comment les structures de contrôles habituelles des autres langages (branchement, boucles, ou même séquences) peuvent être mimées en Caml à l'aide de fonctionnelles. Les méthodes obtenues s'avèrent souvent plus agréables d'utilisation, et avec un peu d'habitude elles viennent plus naturellement à l'esprit.

B.1.1 Un exemple

Supposons qu'on veuille écrire en Caml une fonction qui produise les mêmes effets que les pseudo-instructions Pascal suivantes :

ou les instructions en pseudo-Pascal ?

```
begin z := 1 ;
      while x <= y do
      begin
         z := z * x ;
         x := x + 1
      end
end
```

Clairement ce programme renvoie dans z le produit de tous les entiers de l'intervalle $[x..y]$.

Bien entendu, on pourrait reproduire très exactement ces instructions en Caml en utilisant des références.

beurk !

Programme B.1 Une version fort laide avec références

```
let z = ref 1 and x' = ref x and y' = ref y
in
while !x' <= !y' do
   z := !z * !x' ;
   x' := !x' + 1
done ;
!z
```

Cela fonctionne, certes, mais c'est tout sauf de la programmation fonctionnelle, et je dirai même plus, tout sauf du Caml !

Une première réflexion qu'il faut mener concerne la boucle en cause : il faudra de toutes façons, en style impératif, construire un invariant de boucle pour prouver l'algorithme. Ce faisant, on s'aperçoit que l'état de l'algorithme tout au long des itérations est caractérisé par le triplet (x, y, z).

style impératif ≡ à la Pascal

oui, ici c'est évident !

Considérons les instructions du programme une à une et écrivons, en Caml, la modification correspondante du triplet en question.

La première ligne, `z := 1` correspond à la procédure Caml

```
let f1 (x,y,z) = (x,y,1) ;;
```

De même le corps de la boucle est constitué de deux instructions que l'on peut écrire ainsi :

```
let f2 (x,y,z) = (x,y,z*x) ;;
let f3 (x,y,z) = (x+1,y,z) ;;
let corps = compose f3 f2 ;;
```

où la fonction `compose` est classiquement définie par

```
let compose f g x = f (g x) ;;
```

On aura noté que `f1`, `f2`, `f3` et `corps` sont du type `int*int*int -> int*int*int`.

Remarquons encore qu'on aurait pu écrire directement

```
let corps (x,y,z) = (x+1,y,z*x) ;;
```

Intéressons-nous maintenant au test, qu'il est facile d'écrire :

```
let test (x,y,z) = x <= y ;;
```

Reste à écrire un équivalent du `while` :

```
let rec boucle (x,y,z) =
    if test (x,y,z) then boucle (corps (x,y,z))
                    else (x,y,z) ;;
```

Finalement `let prog = compose boucle f1` définit tout simplement le programme complet, ou, en développant et simplifiant :

Programme B.2 Une version fonctionnelle

```
let prog (x,y,z) =
     let rec boucle (x,y,z) =
          if x <= y then boucle (x+1,y,z*x)
                    else (x,y,z)
     in
     boucle (x,y,1) ;;
```

En relisant le code obtenu, on s'aperçoit aisément qu'il n'est pas utile de faire figurer `y` dans la définition de `boucle`, ce qui permettrait de simplifier encore un peu notre programme. En outre, il est plus naturel de demander au programme de ne rendre que la valeur finale de `z`, et de n'attendre en argument que les valeurs de `x` et `y`. Enfin, on pourrait aussi tout réécrire sous forme curryfiée. On trouve alors la version définitive :

Programme B.3 La version finale

```
let prog x y =
     let rec boucle x z =
          if x <= y then boucle (x+1) (z*x)
          else z
     in boucle x 1 ;;
```

Cette dernière version paraît assez naturelle, je trouve, et avec de l'habitude c'est celle que l'on écrit directement.

B.1.2 Retour au problème général : la séquence

Comme nous venons de le voir, nous allons simuler les structures habituelles à l'aide de fonctions Caml qui modifieront une variable `état` qui contient l'ensemble des informations à maintenir tout le long du programme. Dans notre exemple, `état` était le triplet $(\mathtt{x}, \mathtt{y}, \mathtt{z})$.

La séquence est facile à écrire : si `f1` et `f2` traduisent l'effet sur la variable `état` des instructions `C1` et `C2`, alors c'est bien entendu `compose f2 f1` qui traduit la séquence `C1;C2`.

On notera pour simplifier `C1` $\rightsquigarrow$ `f1`, `C2` $\rightsquigarrow$ `f2`, puis `C1;C2` $\rightsquigarrow$ `compose f2 f1`, voire tout simplement ; $\rightsquigarrow$ `compose`.

une fonctionnelle est une fonction qui opère sur des fonctions

On dira ainsi que la structure de contrôle que constitue la séquence est implémentée par la fonctionnelle `compose`.

B.1.3 Boucle `while`

Considérons maintenant le problème de la structure de contrôle `while`. Il s'agit d'écrire la fonctionnelle `bouclewhile` correspondante. Elle attendra en arguments des fonctions, bien sûr, à savoir `test` et `corps`.

Finalement, si la variable `état` est du type Caml α, notre fonctionnelle `bouclewhile` a pour type

$$(\alpha \to \texttt{bool}) \to (\alpha \to \alpha) \to \alpha \to \alpha,$$

puisque $\alpha \to$ `bool` est le type de `test`, et $\alpha \to \alpha$ le type de chaque instruction, donc de `corps` mais aussi du résultat de `bouclewhile test corps`.

L'exemple étudié plus haut nous guide vers la définition de `bouclewhile` :

```
let bouclewhile test corps =
      let rec boucle état = if (test état) then boucle (corps état)
                                           else état
      in boucle ;;
```

ou, plus simplement encore :

```
let rec bouclewhile test corps état =
      if (test état) then bouclewhile test corps (corps état)
                     else état ;;
```

B.1.4 Boucle de comptage

Dans le même ordre d'idée, écrivons maintenant la fonctionnelle qui correspond à une boucle de comptage, qu'on pourrait écrire de façon impérative par

```
répéter n fois
begin
<corps de la boucle>
end
```

Ainsi notre fonctionnelle aura cette fois, avec les conventions précédentes, un type égal à

$$\texttt{int} \to (\alpha \to \alpha) \to \alpha \to \alpha.$$

Elle n'est pas difficile à écrire :

```
let rec répéter n corps état =
      if n < 1 then état
               else répéter (n-1) corps (corps état) ;;
```

B.2 Gestion de suites infinies

Dans cette section, nous allons étudier le moyen de représenter en Caml des structures infinies, au moyen de ce qu'on appelle d'habitude l'évaluation paresseuse.

en anglais, lazy evaluation

B.2.1 L'idée de l'évaluation paresseuse

On ne peut évidemment pas — en quelque langage qu'on travaille — conserver physiquement une information de taille infinie. On va donc plutôt conserver l'information nécessaire au calcul d'un nombre infini de termes.

du hard

Plus précisément, voulant implémenter des listes infinies dénombrables, ce que nous appellerons des *suites*, nous utiliserons le type suivant :

```
type 'a suite_infinie = Nil | Cellule of (unit -> 'a * 'a suite_infinie) ;;
```

`Nil` représente bien sûr la suite vide. Sinon, une suite est de la forme `Cellule(f)` où `f` est une fonction sans argument dont *l'évaluation* fournit le couple `(tête,queue)` qui représente la liste. Ainsi, ce ne sont pas les valeurs qui sont conservées, mais plutôt les méthodes nécessaires à leur calcul. Si l'on veut accéder à la dixième valeur de la suite, il faudra passer par 10 évaluations successives.

On écrit sans difficulté les fonctions d'accès aux suites :

```
let cons x l =
    let f () = (x,l)
    in Cellule f  ;;

let tête = function
      Nil -> raise Suite_Vide
    | Cellule f -> match f() with x,_ -> x ;;

let queue = function
      Nil -> raise Suite_Vide
    | Cellule f -> match f() with _,q -> q ;;

let est_vide = function
      Nil -> true
    | _ -> false ;;
```

où on a bien sûr défini l'exception `Suite_Vide`.

On écrit également une fonction `force`, qui prend en arguments un entier `n` et une suite et qui rend un couple formé de la liste (une vraie liste Caml) des `n` premiers éléments de la suite et du reste de la suite, ce qui forme une suite infinie.

C'est une fonction récursive simple qui s'écrit :

```
let rec force n l = match n,l with
      0,l -> [],l
    | n,Nil -> raise Suite_Vide
    | n,Cellule f ->
           match f() with x,q ->   let liste,reste = force (n-1) q
                                   in (x :: liste),reste ;;
```

À partir de là, on peut, pour la facilité d'utilisation, définir :

```
let premiers n l = match force n l with liste,_ -> liste ;;
let reste n l = match force n l with _,r -> r ;;
```

B.2.2 Quelques suites simples

On écrit facilement la suite des entiers supérieurs ou égaux à `n` :

```
let rec à_partir_de n = let f () = n,(à_partir_de (n+1)) in Cellule f ;;
```

Alors on a tout simplement : `let N = à_partir_de 0 ;;` qui représente bien *tout* $\mathbb{N}$!

On pourrait aussi définir une suite constante égale à 1 par :

```
let suite_de_1 = let rec f() = 1,(Cellule f) in Cellule f ;;
```

Si en revanche on souhaite implémenter pour les suites un genre de fonction `filter` comme on l'a fait ci-dessus pour les listes, il faudra être plus prudent, et ne pas exécuter la récursion complète (elle serait infinie), mais la retarder grâce à l'utilisation d'une fonction locale.

les Américains utilisent le verbe delay

Voici le code obtenu :

```
let rec filtre prédicat = function
      Nil -> Nil
    | Cellule f -> match f() with x,q ->
        if (prédicat x) then
            let g () = x,(filtre prédicat q)
            in Cellule g
        else filtre prédicat q ;;
```

Par exemple, écrivant `let non_multiple a b = (b mod a) <> 0 ;;`, on obtiendra les entiers naturels impairs par l'instruction

```
let entiers_impairs = filtre (non_multiple 2) N ;;
```

B.2.3 La suite des nombres premiers

On pourrait s'amuser à définir les opérations courantes sur les suites : addition, produit de Cauchy, dérivation, etc. Nous allons plutôt, car cela semble plus amusant, définir la suite des nombres premiers, grâce au célèbre crible d'Ératosthène. En fait nous avons fait le plus difficile. Il ne reste plus qu'à écrire la fonction `crible` elle-même.

```
let élimine x l = filtre (non_multiple x) l ;;
let rec crible = function
      Nil -> raise Suite_Vide
    | Cellule f -> match f() with x,q ->
            let g() = x,(crible (élimine x q))
            in Cellule g ;;
```

Pour terminer il suffit d'écrire `let nombres_premiers = crible (à_partir_de 2) ;;`

Programme B.4 Utilisation des suites infinies

```
1 >       Caml Light version 0.6
2
3 #include "Suites_infinies.ml";;
4 ...
5 - : unit = ()
6 #premiers 10 entiers_naturels;;
7 - : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
8 #premiers 20 nombres_premiers;;
9 - : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41;
10                43; 47; 53; 59; 61; 67; 71]
```

Programme B.5 Implémentation des suites infinies

```
type 'a suite_infinie = Nil | Cellule of (unit -> 'a * 'a suite_infinie);;

exception Suite_Vide;;

let cons x l =
    let f () = (x,l)
    in Cellule f ;;

let tête = function
      Nil -> raise Suite_Vide
    | Cellule f -> match f() with x,_ -> x;;

let queue = function
      Nil -> raise Suite_Vide
    | Cellule f -> match f() with _,q -> q;;

let est_vide = function
      Nil -> true
    | _ -> false;;

let rec force n l = match n,l with
      0,l -> [],l
    | n,Nil -> raise Suite_Vide
    | n,Cellule f ->
           match f() with x,q ->   let liste,reste = force (n-1) q
                                   in (x :: liste),reste;;

let premiers n l = match force n l with liste,_ -> liste;;
let reste n l = match force n l with _,r -> r;;

let rec à_partir_de n = let f () = n,(à_partir_de (n+1)) in Cellule f;;

let entiers_naturels = à_partir_de 0;;

let suite_de_1 = let rec f() = 1,(Cellule f) in Cellule f;;

let rec filtre prédicat = function
      Nil -> Nil
    | Cellule f -> match f() with x,q ->
        if (prédicat x) then
            let g () = x,(filtre prédicat q)
            in Cellule g
        else filtre prédicat q;;

let non_multiple a b = (b mod a) <> 0;;

let entiers_impairs = filtre (non_multiple 2) entiers_naturels;;

let élimine x l = filtre (non_multiple x) l;;

let rec crible = function
      Nil -> raise Suite_Vide
    | Cellule f -> match f() with x,q ->
           let g() = x,(crible (élimine x q))
           in Cellule g;;

let nombres_premiers = crible (à_partir_de 2);;
```

Annexe C

Glossaire franco-anglais

à	to	à reculons vers	downto
abandonner	abort	afficher	display
algorithme	algorithm	alors	then
anneau	ring	arbre	tree
arête	edge	avec	with
booléen	boolean	chaîne	string
chemin	path	clé	key
compilateur	compiler	corps (algèbre)	field (algebra)
court	short	couvrant	spanning
créer	create	dans	in
de	of	début	begin
en tant que	as	enregistrement	record
ensemble	set	entier	integer
enveloppe convexe	convex hull	erreur	error
espace vectoriel	vector space	essayer	try
et	and	exponentiel	exponential
extraire	extract	faire	do
fait	done	faux	false
file d'attente	queue	file de priorité	priority queue
filtrage	pattern matching	filtrer	match
fin	end	fonction	function
forêt	forest	fusion	union
graphe	graph	groupe	group
impair	odd	imprimer	print
insérer	insert	interprète	interpreter
jusque	until	langage	language
lien	link	listage	listing
liste	list	lister	list
logarithme	logarithm	longueur	length
matrice	matrix	modifiable	mutable
motif	pattern	nœud	node
nombre premier	prime number	nombre réel	floating point number
non	not	pair	even
ou	or	où	where
paresseux	lazy	partition	partition
perforer	punch	pile	stack
pointeur	pointer	polynôme	polynomial
preuve	proof	programme	program
préfixe	prefix	racine	root
racine carrée	square root	retarder	delay
si	if	sinon	else
soit	let	sommet	vertex
sortir	exit	stopper	stop
structure	structure	supprimer	delete
tant que	while	tas	heap
tri	sort	trier	sort
trouver	find	type	type
union	union	valeur	value
vecteur	vector	vers	to
vide	empty	vrai	true

Annexe D

Glossaire anglo-français

abort	abandonner	algorithm	algorithme
and	et	as	en tant que
begin	début	boolean	booléen
compiler	compilateur	convex hull	enveloppe convexe
create	créer	delay	retarder
delete	supprimer	display	afficher
do	faire	done	fait
downto	à reculons vers	edge	arête
else	sinon	empty	vide
end	fin	error	erreur
even	pair	exit	sortir
exponential	exponentiel	extract	extraire
false	faux	field (algebra)	corps (algèbre)
find	trouver	floating point number	nombre réel
forest	forêt	function	fonction
graph	graphe	group	groupe
heap	tas	if	si
in	dans	insert	insérer
integer	entier	interpreter	interprète
key	clé	language	langage
lazy	paresseux	length	longueur
let	soit	link	lien
list	liste	list	lister
listing	listage	logarithm	logarithme
match	filtrer	matrix	matrice
mutable	modifiable	node	nœud
not	non	odd	impair
of	de	or	ou
partition	partition	path	chemin
pattern	motif	pattern matching	filtrage
pointer	pointeur	polynomial	polynôme
prefix	préfixe	prime number	nombre premier
print	imprimer	priority queue	file de priorité
program	programme	proof	preuve
punch	perforer	queue	file d'attente
record	enregistrement	ring	anneau
root	racine	set	ensemble
short	court	sort	tri
sort	trier	spanning	couvrant
square root	racine carrée	stack	pile
stop	stopper	string	chaîne
structure	structure	then	alors
to	vers	to	à
tree	arbre	true	vrai
try	essayer	type	type
union	fusion	union	union
until	jusque	value	valeur
vector	vecteur	vector space	espace vectoriel
vertex	sommet	where	où
while	tant que	with	avec