TPL2 : recherche d'une solution optimale au taquin "Embouteillage"

Dans ce TP, on veut construire un programme `embroche`[1] qui, étant donné une configuration initiale au jeu de embouteillage, recherche s'il est possible de sortir la voiture horizontale sur la 3ième ligne du plateau à partir de cette configuration initiale et qui dans ce cas calcule une *solution* (c'est-à-dire une liste de coups permettant d'atteindre une configuration gagnante) dont la consommation est minimale (parmi toutes les autres solutions possibles). Concrètement, `embroche` prend comme argument sur la ligne de commande le nom du fichier de la configuration initiale et affiche sur la sortie standard la liste de coups dans le format des commandes de `semba`. Ce programme va utiliser le paquetage `Taquin` décrit en TPL1.

1 Présentation et analyse du problème

La recherche d'une solution optimale au jeu de taquin est un cas particulier de l'algorithme de Dijkstra que vous allez étudier au second semestre. Cet algorithme recherche un plus court chemin dans un graphe orienté à partir d'une origine unique. Cette section présente le principe de cet algorithme dans le cas particulier qui nous intéresse, sans faire référence aux graphes orientés que vous verrez au second semestre.

1.1 Algorithme de recherche d'une solution optimale

On va s'intéresser à l'ensemble des configurations *atteignables* du plateau de jeu. Une configuration *atteignable* est une configuration du plateau dans laquelle on peut arriver en jouant un nombre fini de coups depuis la configuration initiale. On définit aussi la *consommation* d'une configuration atteignable comme la consommation minimale nécessaire pour atteindre cette configuration depuis la configuration initiale. Ainsi, la consommation de la configuration initiale est 0. Voir les exemples de la figure TPL0.1. Par ailleurs, on dit qu'une configuration B est *la voisine* d'une configuration A si et seulement si on peut passer de A à B en jouant 1 coup. Remarquons que sur un taquin, on peut toujours jouer un coup inverse à celui qu'on vient de jouer : la relation "être voisin" est donc symétrique.

L'algorithme procède en deux étapes. Dans la première étape, il explore l'ensemble des configurations atteignables du plateau *par ordre croissant de consommation*. Il s'arrête soit quand il ne trouve plus de nouvelles configurations atteignables, soit à la première configuration gagnante rencontrée. Dans ce dernier cas, on peut alors passer à la deuxième étape : il reconstruit la liste de coups permettant d'arriver à la configuration gagnante.

Concrètement, on introduit les types de données ci-dessous. Le type `ClefDico` est défini comme un synonyme du type abstrait `Taquin.Etat` (l'intérêt d'avoir un synonyme sera expliqué en section 2.2). Le type `Config` correspond à une succession, sous forme de liste chainée, de configurations atteignables qui ont été construites par l'algorithme lors de la première étape. Le type `PositionFile` est un type de pointeur sur un type abstrait `CellListeDC` dont le rôle sera expliqué en section 1.3.

```
subtype ClefDico is Etat ;

type CellListeDC is private ;
type PositionFile is access CellListeDC ;
```

[1]`EmbROCHE` est l'acronyme de *"EMBouteillage : Résolution avec Optimisation de la Consommation Hypothétique d'Essence."* Ici le mot "*hypothétique*" signifie que le programme est en fait paramétré par une fonction `Conso` censée estimer la consommation d'essence du déplacement des véhicules.


```
type CellConfig ;
type Config is access CellConfig ;

type CellConfig is record
   Clef: ClefDico ;
   Conso: Natural ;
   Index: PositionFile := null ;
   Prec: Config := null ;
   CoupPrec: Coup ;
end record ;
```

Une valeur `S` du type `Config` est donc un pointeur vers un enregistrement qui a 5 champs :

- Le champ `Clef` représente l'état du plateau de la configuration `S` qui doit être une configuration atteignable (depuis la configuration initiale).
- Le champ `Conso` est un *majorant* de la consommation de la configuration, c'est-à-dire que `Conso` est supérieur ou égal à la consommation minimale nécessaire pour atteindre `S` depuis la configuration initiale. Lorsque le champ `Conso` correspond bien à la consommation de la configuration (lorsqu'il y a égalité), on dit par la suite qu'il est *définitif*. On va expliquer plus tard dans quelles conditions on sait que le champ `Conso` est définitif.
- Si `S` est la configuration initiale, alors `S.Prec` vaut **null**. Sinon, le champ `Prec` n'est pas **null** et correspond à une configuration voisine de `S` qui a permis d'arriver dans la configuration `S` depuis la configuration initiale. Ainsi `S` est la tête d'une liste chaînée qui permet de remonter à la configuration initiale en suivant les champs `Prec`, de voisine en voisine.
- Si `S` est la configuration initiale, le champ `CoupPrec` n'a aucune signification. Sinon, le champ `CoupPrec` indique le coup par lequel on passe de `S.Prec` à `S`. De plus, si C est le résultat de `Conso(S.CoupPrec)` lorsque le plateau est dans la configuration `S.Prec`, on doit avoir :

  `S.Conso = S.Prec.Conso +` C
- Le rôle du champ `Index` sera expliqué en section 1.3.

Attention, dans la suite, la "consommation d'une configuration" *fait toujours référence à sa* "consommation minimale"*. C'est donc à bien distinguer de la valeur de son champ `Conso` (sauf si l'on sait qu'il est définitif).*

La première étape de l'algorithme va être implémentée ici par la fonction `ChercheGagnant` qui prend en argument le nom de fichier de la configuration initiale, et retourne une valeur de type `Config` correspondant à une configuration gagnante s'il en existe une, et qui retourne **null** sinon.

```
function ChercheGagnant(ConfigInitiale:String) return Config ;
```

Un point crucial garanti par cette fonction est que si la valeur `S` retournée est non nulle, alors `S.Conso` est *définitif* (il correspond bien à la consommation minimale nécessaire pour atteindre `S` à partir de la configuration initiale en entrée). Remarquons que cela garantit en particulier que toutes les configurations accessibles en suivant les champs `Prec` depuis `S` ont aussi un champ `Conso` définitif (sinon, `S.Conso` ne serait pas la consommation !). Bien sûr, cette fonction garantit aussi qu'il n'y a pas de configuration gagnante de consommation strictement plus petite que celle de `S`.

La deuxième étape de l'algorithme va être implémentée ici par la procédure `Joue`. Celle-ci se contente de parcourir à l'envers la liste chaînée dont la tête est retournée par `ChercheGagnant`

en affichant les coups.

```
procedure Joue(S: Config) ;
```

Le bloc principal de `embroche` implémentant l'algorithme est donc très simplement quelque chose comme :

```
   Gagnant: Config ;
begin
   Gagnant:=ChercheGagnant(ConfigInitiale) ;
   if Gagnant /= null then
      Joue(Gagnant) ;
   else
      ....
   end if ;
```

Le gros du travail est donc dans la fonction `ChercheGagnant`. Elle utilise une structure de données particulière, appelée un *dictionnaire*. Ce dictionnaire permet de stocker l'ensemble des configurations rencontrées au cours de l'exploration. Ces configurations sont par la suite dites *connues*. Plus précisément, les procédures et fonctions du dictionnaire permettent :

- de savoir si un état du plateau correspond à une configuration connue, et si oui, laquelle.
- d'ajouter une nouvelle configuration connue dans le dictionnaire, lorsque celle-ci n'est pas déjà présente dans le dictionnaire.

Par ailleurs, la fonction `ChercheGagnant` utilise aussi une autre structure de données, appelée *file de priorités*. Essentiellement, cette file de priorités sert à stocker les configurations connues, dont on n'est pas encore sûr que leur champ `Conso` soit définitif. Plus précisément, l'algorithme nous garantit les trois invariants suivants sur la file :

1. Toute configuration voisine d'une *configuration connue absente de la file* est elle-même une configuration connue (mais éventuellement dans la file). Voir la figure 1.
2. Pour toute configuration connue `S`, la valeur `S.Conso` représente "*la consommation minimale nécessaire pour atteindre cette configuration depuis la configuration initiale, en ne passant que* **par des configurations connues absentes de la file**". Autrement dit, il existe peut-être une liste de coups ayant une consommation inférieure, mais passant par des configurations de la file ou par des configurations encore inconnues.
3. Étant donné deux configurations connues `S1` et `S2` avec `S1` absent de la file et `S2` présent dans la file, alors la consommation de `S2` est supérieure ou égale à `S1.Conso`. (Ainsi, `S1.Conso<=S2.Conso` puisque `S2.Conso` est supérieur ou égal à la consommation de `S2`).

Des invariants qui précèdent, on peut déduire les faits suivants :

1. *Toute configuration atteignable inconnue a une consommation supérieure ou égale au champ `Conso` de n'importe quelle configuration connue absente de la file.*
   En effet, considérons une configuration X ayant la plus petite consommation parmi les configurations atteignables inconnues. Il y a nécessairement une voisine Y de X, qui a une consommation inférieure à celle de X et qui est une configuration connue (sinon, X n'a pas la plus petite consommation parmi les configurations atteignables inconnues). Cette configuration Y est dans la file (sinon X serait une configuration connue d'après l'invariant 1). La consommation de X est supérieure à la consommation de Y qui est elle-même supérieure à toutes les configurations connues absentes de la file (invariant 3). CQFD.
2. *Toute configuration connue absente de la file a un champ `Conso` définitif.*
   En effet, soit `S` une configuration connue absente de la file. Par le point précédent et l'invariant 3, toute suite de coups atteignant `S` qui passe par une configuration qui n'est pas une configuration connue absente de la file, exige nécessairement une consommation supérieure ou égale à `S.Conso`. Donc l'invariant 2 garantit que `S.Conso` correspond bien à la consommation de `S`.
3. *Toute configuration de la file ayant un champ `Conso` minimal parmi les configurations de la file a un champ `Conso` définitif.*
   En effet, appelons `S` une configuration de la file ayant le plus petit champ `Conso`. Comme précédemment toute suite de coups atteignant `S` qui passe par une configuration qui n'est pas une configuration connue absente de la file exige nécessairement une consommation supérieure à `S.Conso`.
4. *Lorsque la file est vide, toutes les configurations atteignables sont connues.* C'est une conséquence directe de l'invariant 1.

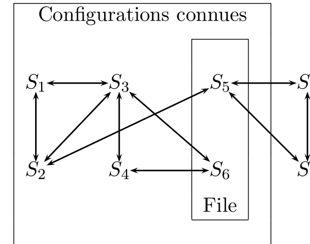


FIG. 1 – Différents types de configurations avec les relations de voisinage. Les configurations S_1 à S_6 sont des configurations connues et les configurations S_5 et S_6 sont dans la file. D'après l'invariant 1, les configurations S_7 et S_8 ne sont voisines que des configurations présentes dans la file.

Ainsi, la fonction `ChercheGagnant` part de la configuration initiale avec un `Conso` à 0, qu'elle enregistre comme unique élément dans le dictionnaire et dans la file. Ensuite, elle répète la séquence suivante qui préserve les invariants décrits ci-dessus :

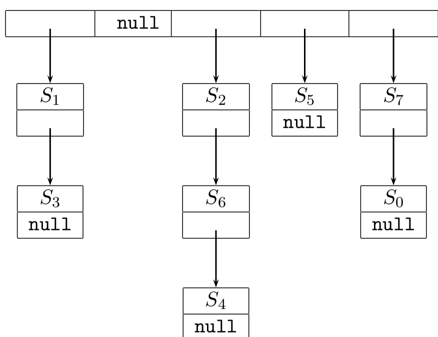
1. Chercher et retirer la configuration `SI` de champ `Conso` minimal dans la file. (En effet, `SI.Conso` est définitif).
2. Si `SI` correspond à une configuration gagnante, on retourne `SI` (on s'arrête donc sur une configuration gagnante dont la consommation est minimale parmi les configurations atteignables).
3. Sinon, on énumère toutes les configurations `SF` voisines de `SI` en effectuant pour chacune :
   - On calcule la consommation C par le coup passant de `SI` à `SF`.
   - Si `SF` n'est pas une configuration connue (du dictionnaire), on l'ajoute dans le dictionnaire et dans la file avec la consommation C+`SI.Conso` (et le champ `Prec` valant `SI`).
   - Si `SF` est une configuration connue, alors on teste si C+`SI.Conso < SF.Conso`. Dans ce cas, il faut mettre `SF.Conso` à jour avec cette nouvelle valeur (le champ `SF.Prec` étant lui aussi modifié avec la valeur `SI`).

Cette séquence est répétée tant que la file n'est pas vide et qu'on n'a pas trouvé une configuration gagnante. Lorsque la file est vide, c'est qu'il n'y a pas de configuration gagnante atteignable.

1.2 Le dictionnaire comme une table de hachage

En général, le rôle d'un dictionnaire est d'enregistrer un ensemble de données (ici les configurations connues) de manière à pouvoir retrouver la donnée associée à une *clé* (ici un état du plateau). Une première idée serait d'implémenter le dictionnaire comme une liste chaînée de configurations. En effet, on a besoin d'une structure dont la taille peut être étendue dynamiquement, car on n'a aucune idée à priori du nombre de configurations que l'on doit stocker. Mais la fonction de recherche dans une telle liste est coûteuse, car dans le pire cas, on doit parcourir toute la liste. Or celle-ci va peut-être contenir plusieurs dizaines de milliers d'éléments. Pour réduire le temps de recherche, on va plutôt utiliser une *table de hachage*.

On suppose donné un entier strictement positif N, et une fonction (dite *de hachage*) h qui associe à chaque clé un nombre entre 0 et $N-1$. La *table de hachage* est un tableau de N listes indicées de 0 à $N-1$, de sorte que pour tout indice i de 0 à $N-1$ et pour toute configuration de clé C présente dans la liste d'indice i, $h(C)$ vaut i. Voir la figure 2.



Dans cet exemple, les S_i représentent des configurations dont les clés sont notées C_i. On a nécessairement :

$$h(C_1) = h(C_3)$$
$$h(C_2) = h(C_6) = h(C_4)$$
$$h(C_7) = h(C_0)$$

FIG. 2 – Une table de hachage pour $N = 5$

Ainsi, pour savoir s'il y a une configuration de clé C dans la table, il suffit de la rechercher dans la liste d'indice $h(C)$. Si la fonction de hachage est uniforme (c'est-à-dire si les indices retournés par h sont équiprobables), on divise donc le temps de recherche par N. Autrement dit, s'il y a M entrées dans la table, le temps de recherche d'une configuration va être en $\Theta(\frac{M}{N})$.

Comme on ne connaît pas nécessairement à l'avance la valeur maximale que va atteindre M au cours de l'exécution du programme, il est intéressant d'augmenter N au fur et à mesure que la valeur de M augmente (il faut alors aussi changer h en conséquence). Par exemple, étant donné une certaine constante K supérieure ou égale à 2, on peut s'arranger pour que $M \leq K \times N$. De cette façon, le temps de recherche est en $\Theta(1)$. Par contre, l'insertion devient plus délicate : lorsqu'on insère une configuration alors que $M = K \times N$, il faut redimensionner la table de hachage (c'est-à-dire, recopier les entrées du tableau courant dans un tableau de taille plus grande). Ce redimensionnement a un coût proportionnel à M. Ceci dit, si la nouvelle taille lors d'un redimensionnement vaut systématiquement K fois l'ancienne taille N (on redimensionne

avec la taille $K \times N$), alors le redimensionnement arrive d'autant moins souvent que M est grand. Pour fixer les idées, prenons un dictionnaire vide dont la taille du tableau N est non-nulle et avec $K = N$ (c'est-à-dire, la valeur de K est égale à la taille initiale du tableau). Ainsi, chaque redimensionnement intervient si une insertion arrive alors que $M = K^i$ avec $i \geq 2$. Le coût de M insertions à partir du dictionnaire vide est donc en $\Theta(M)$, puisque le *coût total des redimensionnements* $\sum_{i=2}^{\log_K(M)} K^i$ est inférieur à $\frac{K^{\log_K(M)+1}}{K-1}$ (i.e. $\frac{K}{K-1} \times M$).

1.3 File de priorités de Dial

Une file de priorités est une structure de données qui représente un ensemble d'éléments (ici des configurations) ordonnés suivant un certain attribut (ici le champ `Conso`) appelé *priorité*. Vous reverrez les principes des files de priorités au second semestre. Ici on étudie une implémentation très dépendante de la façon dont on l'utilise dans `ChercheGagnant`.

On cherche à définir une structure de file de priorités avec un coût relativement efficace sur les opérations suivantes utilisées dans `ChercheGagnant` :

- extraction de la configuration dont le champ `Conso` est minimal de la file.
- insertion d'une nouvelle configuration dans la file.
- modification d'un champ `Conso` d'une configuration déjà présente dans la file.

Soit `ConsoMax` la consommation maximale d'un unique coup. On peut montrer que l'algorithme donné pour `ChercheGagnant` préserve l'invariant ci-dessous, appelé *invariant de Dial* dans la suite : *la différence de valeur pour le champ `Conso` entre deux configurations quelconque de la file est au plus `ConsoMax`*. En effet, si `SI` est la configuration de champ `Conso` minimal qui est extraite de la file, alors les configurations voisines de `SI` qui sont ajoutées dans la file ou dont le champ `Conso` est modifié, le sont avec un champ `Conso` compris entre `SI.Conso` et `SI.Conso+ConsoMax`.

Cet invariant de Dial permet une implémentation de la file qui est très efficace lorsque `ConsoMax` est relativement faible. La file est codée comme un tableau, appelée `File` ci-dessous, dont les éléments sont des listes de configurations, telles que toutes les configurations d'une même liste d'indice `I` ont un champ `Conso` égal à la même valeur `C` modulo la taille de la file (c'est-à-dire, `I = C mod File'Length`) :

```
subtype Indice is Integer range 0..ConsoMax ;
File: array(Indice) of ListeDC ;
```

Ci-dessus le type `ListeDC` est un type représentant des listes *doublement chaînées* dont les éléments sont des configurations. L'intérêt de cette structure de données est qu'on peut retirer une cellule du chaînage à coût constant dès qu'on a un pointeur sur cette cellule (alors que pour un chaînage simple il faut connaître un pointeur sur la cellule précédente dans le chaînage).

En effet, pour modifier le champ `Conso` d'une configuration `S` de la file, on va devoir extraire cette configuration de la liste où elle se trouve pour l'insérer dans une autre liste de la file : la liste `File(S.Conso mod File'Length)` si `S.Conso` est la nouvelle valeur qui remplace l'ancienne.

Pour réaliser cette double opération sur les listes (extraction et insertion) à coût constant, on va donc conserver dans chaque configuration `S` de la file un pointeur `S.Index` sur la cellule de la liste qui contient `S`. C'est le rôle du champ `Index` de type `PositionFile` introduit précédemment dans le type `CellConfig`. Un champ `S.Index` valant **null** indiquera que `S` n'est pas dans la file.

Ainsi, le type `PositionFile` représente le type des pointeurs sur les cellules des listes doublement chaînées de type `CellListeDC` :

```
type PositionFile is access CellListeDC ;

type CellListeDC is record
```

```
   Val: Config ;
   Prec, Suiv: PositionFile ;
end record ;

-- Invariant:
--  pour S: Config, si S.Index /= null alors S.Index.Val = S.
```

L'implémentation attendue des listes doublement chaînées est précisée en section 1.4.

Les principes énoncés juste ci-dessus permettent donc de coder les opérations d'insertion ou de modification dans la file à coût constant. Il reste maintenant à examiner comment extraire le minimum. On suggère d'utiliser les variables globales suivantes dans l'implémentation du paquetage `FilePrio` :

```
NbElems: Natural ;     -- nombre d'éléments présents dans la file.
IndiceMin: Indice ;    -- indice de la liste de priorité minimum.
```

On introduira l'invariant suivant : si `NbElems` est non nul, alors `File(IndiceMin)` est une liste non vide qui contient les configurations de champ `Conso` minimal dans la file. Dans le pire cas (assez rare à priori), pour mettre-à-jour `IndiceMin` lors d'une extraction, il faudra donc parcourir circulairement l'ensemble du tableau pour trouver la première liste non vide, ce qui coûte donc de l'ordre de `ConsoMax` comparaisons de pointeurs. Dans le meilleur cas (assez fréquent à priori), `IndiceMin` reste inchangé : cela se fait à coût constant.

Au final, dans cette implémentation, extraire la configuration de `Conso` minimal peut coûter plus cher qu'une insertion ou qu'une modification, mais arrive à priori moins souvent.

1.4 Listes doublement chaînées circulaires

Comme vu précédemment, le type `PositionFile` représente le type des pointeurs sur les cellules des listes doublement chaînées. Le paquetage des listes doit garantir que tout pointeur `P: PositionFile` correspondant à un pointeur alloué du tas vérifie **l'invariant suivant** : "*`P.Suiv` est un pointeur alloué tel que `P.Suiv.Prec=P`.*"

Pour programmer sur les listes doublement chaînées, on s'impose la discipline suivante :

> *Ne jamais modifier directement les champs `Suiv` et `Prec` des positions. Passer plutôt par l'intermédiaire de la procédure `MetSuiv` ci-dessous.*

```
procedure MetSuiv(Srce,Dest:PositionFile) is
--requiert: Srce et Dest alloués
begin
   Srce.Suiv := Dest ;
   Dest.Prec := Srce ;
end ;
```

En effet, l'utilisation de cette procédure a l'intérêt de rendre le code plus lisible et d'établir l'invariant pour le pointeur `Srce`.

Une liste doublement chaînée est simplement codée par une cellule spéciale du chaînage appelée par la suite *sentinelle*. Le type `ListeDC` est simplement défini comme un renommage de `PositionFile` ce qui permet de bien distinguer par typage la sentinelle des autres cellules.

```
type ListeDC is new PositionFile ;
```

Concrètement, pour toute liste `L: ListeDC` allouée, `L.Val` n'a aucune signification et peut être quelconque. De plus, si `L.Suiv=PositionFile(L)`[2] alors la liste est vide. Sinon, `L.Suiv` et `L.Prec` sont deux pointeurs alloués qui représentent respectivement la **tête** et la **queue** de la liste. Voir la figure 3.

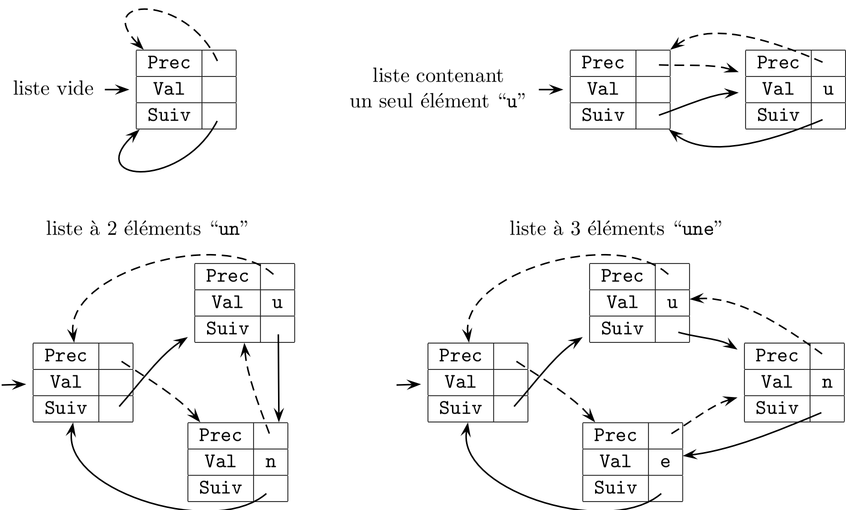


FIG. 3 – Exemples de listes doublement chaînées circulaires

La régularité de cette structure de données a un gros avantage : le code réalisant l'insertion ou la suppression d'un élément de la liste dans le cas général fonctionne aussi sur les cas particuliers (liste vide, liste réduite à un élément, etc). On peut donc se passer de prévoir ces cas particuliers et écrire ainsi un code court et efficace dans le cas général.

1.5 Politique de gestion du tas

Le programme `embroche` explore les configurations atteignables pour une configuration initiale donnée puis s'arrête (après avoir affiché son résultat). Ce n'est donc pas très grave s'il ne rend pas la mémoire du tas inutilisée au système d'exploitation dès qu'il le peut : sur des PCs modernes, il y a peu de risque qu'il se trouve à court de mémoire (à moins d'une programmation vraiment désastreuse). De toute façon, le système d'exploitation récupérera la mémoire à la fin de son exécution. En fait, si on ne cherche pas à récupérer toute la mémoire inutile, `embroche` va être plus simple à programmer, plus sûr et plus rapide : récupérer la mémoire au fur et à mesure de l'exécution, ça prend du temps et c'est une source de bogue. En particulier, à la fin de l'exécution de `ChercheGagnant`, il n'est pas utile de libérer l'espace mémoire occupé par les cellules des listes de type `Config` qui ne font pas partie de la solution optimale, la table de hachage, et la file. En matière de gestion mémoire, on se limitera donc ici à :

- libérer le tableau inutilisé lors du redimensionnement.
- éviter d'allouer des cellules inutilement dans la table de hachage ou dans les listes doublement chaînées.
- libérer la cellule devenue inutile lors d'une extraction d'une liste doublement chaînée (cela n'est ni difficile, ni vraiment coûteux).

[2]Ici, `PositionFile(L)` convertit `L` avec le type `PositionFile`. Mais la valeur du pointeur ne change pas à l'exécution. Cette conversion de type est nécessaire car l'opérateur "=" requiert deux opérandes de même type.

Remarquons que le problème serait différent si la recherche d'une solution optimale était intégrée comme une commande de `semba`. Comme ce dernier est un programme interactif dont l'arrêt est déterminé par l'utilisateur, il est important que la mémoire du programme ne grossisse pas de façon inutile et incontrôlée. Pour récupérer la mémoire des listes de type `Config` au cours des calculs, on pourrait par exemple s'inspirer des listes avec partage de suffixes de l'exo P17.

2 Travail demandé

Il faudra déposer le CR en y joignant le source des programmes à réaliser dans le casier du groupe de TD d'un membre de l'équipe (près de la salle E001 au rez-de-chaussée de l'Ensimag) **avant le vendredi 17 décembre à 14h**. Si tous les membres de l'équipe n'appartiennent pas au même groupe de TD, ils déposent leurs documents dans **un seul** des casiers. Merci d'indiquer au début de chaque document, le numéro d'équipe sur `Teide`, puis le nom et le numéro de groupe de chacun des membres de l'équipe. Par ailleurs, chaque équipe doit aussi déposer sur `Teide` (avant cette même date) ses sources Ada et ses fichiers de tests. Sinon, le barême et les consignes pour le compte-rendu sont identiques au TPL1. Voilà juste quelques conseils supplémentaires pour rédiger un bon compte-rendu :

- il faut des informations synthétiques et pertinentes pour que le relecteur puisse comprendre rapidement le contenu des fichiers fournis (objectif des fichiers de tests, description des fonctions compliqués et des variables importantes pour le code).
- il ne faut pas d'informations factuelles : pas de réflexions personnelles ou/et non-vérifiées qui bruitent le contenu.
- il ne faut pas de détails sur la démarche complète, uniquement sur le résultat obtenu. Typiquement, l'ensemble des tests effectués fait partie du "résultat" (ça permet au relecteur d'évaluer la confiance qu'on peut avoir dans le code produit) alors que la liste des bogues que vous avez corrigés n'a aucun intérêt... Par contre, on veut connaître la liste des bogues détectés mais non corrigés.

La section 2.1 vous donne deux exemples de compte-rendu du TPL1, dont l'un est mal rédigé, mais sans vous préciser lequel c'est : c'est à vous de trouver !

2.1 Paquetage `Taquin` fourni

Comme implémentation du paquetage décrit en TPL1, vous *pouvez* réutiliser votre propre implémentation (sauf par exemple, si elle n'est pas fonctionnelle), mais vous *devez* aussi utiliser une des deux implémentations fournies ci-dessous. Ces deux implémentations sont issues de rendus d'étudiants sur Teide pour le TPL1. En fait, ces implémentations ont été remaniées par les enseignants (pour en particulier les rendre anonymes), mais elles sont assez caractéristiques de ce qu'une promo peut rendre aux enseignants. Parmi ces deux implémentations, il y en a une qui est "*bien*" sans être parfaite. L'autre est plutôt mauvaise. Pour les identifier, il faut s'aider des comptes-rendus associés (fichier `rapport.pdf` dans les archives). Au niveau des comptes-rendus, il y en a aussi *correct* (sans être excellent) alors que l'autre n'est pas très bon. Attention, le compte-rendu correct n'est pas forcément associé à la meilleure implémentation. En fait, le compte-rendu correct vous aide à évaluer rapidement le code associé, alors que la lecture de l'autre vous fait perdre du temps...

- version "Toto" [sur le kiosk]
- version "Titi" [sur le kiosk]

Indiquez dans votre compte-rendu la version que vous utilisez et pourquoi cette version est meilleure que l'autre (en une ou deux phrases).

2.2 Implémentation de la table de hachage

Il est souhaitable de tester que votre table de hachage fonctionne indépendamment du plateau et de l'algorithme de recherche de solution. Pour faciliter cette tâche, on a défini le type `ClefDico` des éléments manipulés par la table comme un synonyme de `Taquin.Etat`. Ainsi, pour faire les tests, il suffit de remplacer la ligne

```
subtype ClefDico is Etat ;
```

par

```
subtype ClefDico is Natural ;
```

Concrètement, on vous fournit les fichiers suivants :

- `configs.ads` [sur le kiosk] : interface du paquetage `Configs` qui définit les types `ClefDico` et `Config`.
- `configs.adb` [sur le kiosk] : implémentation de l'interface ci-dessus.
- `tablehach.ads` [sur le kiosk] : interface de la table de hachage que vous devez réaliser.
- `testtable.adb` [sur le kiosk] : programme pour vous permettre de vérifier que vous avez suffisamment testé votre table de hachage avant de passer à la suite. **Pour compiler, ce fichier requiert** que `ClefDico` soit défini comme un synonyme `Natural`.

Le travail de cette sous-tâche consiste à compléter le fichier fourni `tablehach.adb` [sur le kiosk]. Ce fichier définit notamment le type de données Ada de la table (variable appelée `Dico` dans le fichier fourni).

Votre implémentation doit fonctionner indépendamment de la définition de `ClefDico` du paquetage `Configs`. Pour cela, vous devez utiliser la fonction `Hache` définie dans ce paquetage (le code de cette fonction est choisi via le mécanisme de surcharge par le compilateur dans le fichier `configs.adb`). La fonction h décrite dans à la section 1.2 sera ici implémentée par "$C \mapsto \texttt{Hache}(C) \bmod N$" où N est la taille courante du tableau `Dico`.

Le fichier `tablehach.adb` fourni par les enseignants donne quelques indications pour implémenter le redimensionnement de la table. Ceci dit, dans un premier temps, on pourra implémenter une version de la table de hachage dans laquelle la taille de `Dico` est constante.

Outre, les procédures et fonctions de manipulation de la table nécessaires à l'algorithme de recherche de solution, on vous demande d'implémenter la fonction `Stats` qui permet d'observer les performances du hachage dans la table :

```
type TabNat is array(Natural range <>) of Natural ;
function Stats return TabNat ;
```

Cette fonction retourne un tableau `T` tel que `T'First=0` et `T'Last` est la longueur maximale des listes actuelles de la table. Par ailleurs, elle garantit que pour tout `I`, `T(I)` est le nombre de listes de longueur `I` dans la table. Ainsi, le nombre M de configurations présentes dans la table (retourné par la fonction `NbConfigs`) vérifie

$$M = \Sigma_{i=0}^{\texttt{T'Last}} i.\texttt{T}(i)$$

Écrivez vos propres fichiers de tests pour déboguer le paquetage `TableHach` en utilisant la technique prévue ci-dessus pour `testtable`. Quand vous avez la certitude que votre paquetage fonctionne, testez-le en exécutant le programme `testtable`. Comparez les temps d'exécutions :

1. sans redimensionnement dynamique
2. avec redimensionnement dynamique

2.3 Implémentation des listes doublement chaînées

Il faut implémenter le sous-paquetage de `Configs` appelé `Configs.ListesDC` dont l'interface est donnée dans `configs-listesdc.ads` [sur le kiosk]. Pour cela, vous devez compléter le fichier `configs-listesdc.adb` [sur le kiosk]. Techniquement, ce mécanisme de sous-paquetage permet de masquer la représentation du type `CellListeDC` (via un type privé) dans `Configs` pour tous les autres paquetages, sauf pour `Configs.ListesDC` qui en a besoin.

Le fichier fourni `exempletestlistesdc.adb` [sur le kiosk] donne un exemple d'utilisation du paquetage et précise en commentaire le comportement attendu. Vous pouvez vous inspirez de ce fichier pour réaliser vos propres tests.

2.4 Implémentation de la file de priorités

Il faut implémenter le paquetage `FilePrio` dont l'interface est donnée dans `fileprio.ads` [sur le kiosk] en complétant le fichier fourni `fileprio.adb` [sur le kiosk].

On vous demande ici d'implémenter un mécanisme permettant de détecter si votre implémentation de `ChercheGagnant` ne respecte pas l'invariant de Dial (il s'agit donc d'un mécanisme qui vous aidera à déboguer `ChercheGagnant`). Techniquement, lorsque la constante booléenne `Defensif` vaut `True`, votre implémentation du paquetage `FilePrio` doit permettre de vérifier si la précondition de la procédure `RePositionne` est violée. Cette précondition doit être en effet vérifiée par `ChercheGagnant`, et elle garantit l'invariant de Dial. Précisons ici que le compilateur `gnatmake` propage les constantes et optimise le code exécutable en conséquence : les tests sur les constantes booléennes sont donc fait à la compilation et non pas à l'exécution.

Le fichier fourni `exempletestfileprio.adb` [sur le kiosk] donne un exemple d'utilisation du paquetage et précise en commentaire le comportement attendu. Vous pouvez vous inspirez de ce fichier pour réaliser vos propres tests. Il faut en particulier vérifier que le mécanisme de vérification défensive déclenche bien l'exception attendue quand il le faut (et uniquement quand il le faut).

2.5 Implémentation de `embroche`

Sur la ligne de commande, le programme `embroche` prend des arguments de la forme suivante : un nom de fichier `.emb` (fichier de configuration du taquin *Embouteillage* décrit en section TPL0.2.1), éventuellement précédé d'une option "`-n`", "`-t`", ou "`-v`".

```
./embroche [-n|-t|-v] fich.emb
```

Sans option, la sortie du programme `embroche` est une liste de commandes du programme `semba` qui joue la solution optimale. Par exemple, pour jouer la solution optimale sur `n04.emb`, on lance :

```
./embroche n04.emb | ./semba
```

Sur une configuration initiale qui n'a pas de solution, `embroche` affichera une liste de coups vide, non terminée par `q` de sorte que `semba` va provoquer l'erreur `ADA.IO_EXCEPTIONS.END_ERROR`.

L'option `-n` affiche juste le nombre de coups de la solution optimale (et rien s'il n'y a pas de solution). L'option `-t` affiche en plus du nombre de coups, des informations sur l'utilisation de la table. L'option `-v` simule la partie gagnante puis affiche les mêmes infos que `-t`.

Les enseignants fournissent ici un paquetage `SolJeu` qui s'occupe de gérer ces options et de l'affichage qu'elles requièrent.


```
-- Paquetage qui effectue l'analyse de la ligne de commandes
-- pour le programme "embre"

-- Offre une interface pour "afficher" la liste de coups
-- (en fonction de la ligne de commande)

with Taquin; use Taquin ;

package SolJeu is

   function ConfigInitiale return String ;
   -- retourne le nom du fichier sur la ligne de commande

   procedure InitJeu(Gagnant: Boolean) ;
   -- démarre l'énumération des coups.
   -- requiert Gagnant=True ssi il existe une solution gagnante.

   procedure Joue(C: Coup) ;
   -- requiert: il y a eu un appel à InitJeu(True)
   -- affichage du coup "C"
   -- (en fonction des options de la ligne de commande)

   procedure QuitteJeu ;
   -- requiert: il y a eu un appel à InitJeu.
   -- termine l'affichage de la liste de coups.

end SolJeu ;
```

Par exemple, pour jouer une solution qui comporte les coups `C1`, `C2`, `C3`, on exécute la séquence :

```
InitJeu(True) ;
Joue(C1) ;
Joue(C2) ;
Joue(C3) ;
QuitteJeu ;
```

Pour une configuration sans solution, on exécute :

```
InitJeu(False) ;
QuitteJeu ;
```

A faire dans cette sous-tâche :
- récupérer l'interface `soljeu.ads` [sur le kiosk] et l'implémentation `soljeu.adb` [sur le kiosk] du paquetage `SolJeu`.
- compléter le fichier `embroche.adb` [sur le kiosk].

2.6 Tests d'intégration et expérimentations

Lorsque votre programme `embroche` semble au point, il est suggéré de réaliser les expérimentations ci-dessous. Éventuellement, vous pouvez écrire un script `bash` vous permettant de lancer `embroche` successivement sur un ensemble de fichiers (avec des options adéquate à `embroche`). Inutile de faire un script complexe avec des tas d'options, on pourra se contenter d'un script qu'on modifie éventuellement à la main pour l'adapter à chacune des expérimentations.
- Tester le programme `embroche` sur l'ensemble des fichiers de configurations fournis par les enseignants.

- Comparer les résultats obtenus (option `-n` notamment) pour différentes implémentations de la fonction `ParamJeu.Conso` (voir celles suggérées dans le fichier `paramjeu.adb` fourni).
- Observer le comportement de la table de hachage pour les différentes valeurs de `RedimC` suggérées dans `tablehach.ads`. On utilisera l'option `-t` de `embroche`. Une brève explication du phénomène observé dans le compte-rendu sera la bienvenue.
- Mesurer la différence de performance obtenue sur l'ensemble des configurations lorsqu'on réduit la table de hachage à une simple liste chaînée (c'est-à-dire en fixant `RedimC` à `1` et en mettant en commentaire le mécanisme de redimensionnement).
- Si vous avez utilisé votre implémentation de `Taquin` au cours du TPL2, identifiez la "bonne" implémentation fournie en 2.1, et faites repasser vos tests dessus (vous pouvez aussi mesurer la différence de coût en temps...)