DS 5

Informatique MP2I

Julien Reichert

Toutes les questions de programmation sont à résoudre dans le langage précisé dans la question ou dans le titre du problème, en fonction de la situation.

Questions de cours ou d'application directe du cours

Question de cours 1 : Dans un arbre bicolore, une proposition du cours annonce les deux inégalités $h \leq 2k$ et $n \geq 2^k - 1$. Rappeler quelles sont les variables h, k et n, prouver la formule et en donner une conséquence essentielle.

Question de cours 2 : Rappeler au moins une implémentation possible d'un tas (en expliquant brièvement les opérations élémentaires) afin que l'ensemble des tas soit équilibré.

Question de cours 3 : Définir la notion de matrice d'adjacence.

Question de cours 4 : Définir la notion de fermeture transitive d'un graphe.

Question de cours 5 : Combien y a-t-il d'arcs au maximum dans un graphe orienté sans circuit à n sommets ?

Question de cours 6 : Donner un exemple de graphe dans lequel il existe un chemin optimal en longueur mais pas de chemin optimal en valeur entre deux sommets (justifier brièvement).

Question de cours 7 : Proposer un problème dont la résolution repose sur un parcours de graphe dont le résultat donne presque immédiatement la réponse au problème (justifier brièvement).

Question de cours 8 : Proposer une analyse comparative des algorithmes de Dijkstra, Floyd-Warshall et Bellman-Ford. Il s'agit de donner les avantages et inconvénients des uns et des autres. Inutile de dépasser une dizaine de lignes, et on pourra répondre sous la forme d'un tableau.

Exercices issus des TD et TP... avec adaptations

Exercice T1 : On donne le type suivant : `type aea = Noeud of char * aea * aea | Feuille of int` (le caractère est un opérateur arithmétique). Écrire en OCaml une fonction de conversion vers le type `aea` d'une expression arithmétique en notation polonaise inversée (l'opérateur après ses deux opérandes) donnée en tant que liste de jetons, ces derniers étant des chaînes de caractères contenant soit un entier soit un opérateur.

Exercice T2 : Réaliser une structure de tas-min en C avec les opérations élémentaires de création d'un tas vide, d'ajout d'une valeur et d'extraction de la racine. En particulier, on n'aura pas d'opération de modification d'une valeur.

Exercice T3 : Avec la structure précédente, écrire en C l'algorithme du tri par tas. On construira et on renverra la version triée d'un tableau en argument (la taille est évidemment en argument aussi, comme d'habitude).

Exercice T4 : Écrire en OCaml une fonction qui prend en argument un graphe orienté représenté par liste d'adjacence et qui détermine s'il admet un chemin eulérien (il n'est pas nécessaire de le construire si oui).

Exercices

Exercice 1 : On considère une structure de trie optimisée pour que les descendants d'un nœud soient mis dans un ABR (version simple sans optimisation de la hauteur) plutôt qu'une liste. Écrire en OCaml les opérations élémentaires de test d'appartenance d'un mot à un tel trie et d'insertion d'un mot dans un tel trie. Il faudra bien réfléchir à la création des types associés avant de se lancer, et on pourra se demander pourquoi les opérations élémentaires sur de tels ABR ne sont pas demandées au préalable...

Exercice 2 : Écrire en C une fonction prenant en argument un graphe orienté représenté en tant que structure ayant comme champs un pointeur vers les sommets (chaînes de caractères), sa taille, un pointeur vers les arcs (qui sont des pointeurs de sommets de taille deux) et sa taille, et déterminant si le graphe peut correspondre au graphe d'une bijection.

Problème 1 : Codage de Prüfer [intégralement en C].

Dans ce problème, on fait l'hypothèse que les sommets des graphes non orientés que nous considérons sont les premiers entiers naturels. La représentation utilisée sera la liste d'adjacence :

```
struct graphe_non_oriente { int nbs; int** liste_adj; };
typedef struct graphe_non_oriente gno;
```

Chaque liste d'adjacence donne les sommets voisins d'un sommet (de manière redondante) et terminera par un `-1` en guise de sentinelle, pour permettre de connaître le nombre de voisins.

Question P1.1 : Rappeler la proposition du cours au sujet des conditions auxquelles un graphe non orienté non vide est un arbre.

Question P1.2 : En déduire une fonction prenant en argument un graphe non orienté et déterminant s'il s'agit d'un arbre. On pourra découper le travail en plusieurs fonctions (conseil).

Le codage de Prüfer d'un arbre-graphe sous les hypothèses précédentes est une méthode de représentation unique et condensée de cet arbre.

La construction du codage de Prüfer reprend l'algorithme suivant (le style présenté est l'indentation significative comme en Python), où une feuille est définie comme un sommet de degré un :

```
Créer une séquence vide P
Tant qu'il reste strictement plus de deux sommets dans l'arbre
  Soit f la feuille d'étiquette minimale
  Ajouter à P (en fin de séquence) l'étiquette du voisin de f
  Retirer f de l'arbre
Renvoyer P
```

Le travail va être décomposé en plusieurs questions.

Dans la suite, on assimilera un sommet du graphe à son étiquette, et le retrait d'un sommet se résumera à une suppression d'arête, afin de ne pas perturber l'indexation de la liste d'adjacence.

Question P1.3 : Créer une fonction prenant en argument un graphe et en renvoyant une copie.

Question P1.4 : Créer une fonction prenant en argument un graphe et un entier et déterminant si l'entier représente un sommet qui est une feuille du graphe. On pourra supposer sans le vérifier que l'entier est un indice légitime de la liste d'adjacence.

Question P1.5 : Créer une fonction prenant en argument un graphe et un entier et retirant le sommet du graphe au sens de la précision ci-avant. Le graphe sera en particulier muté.

Question P1.6 : Créer une fonction prenant en argument un graphe et renvoyant son codage de Prüfer sous forme d'un tableau d'entiers.

Pour la réciproque, deux algorithmes sont proposés (merci Wikipedia...).

Premier algorithme, prenant en entrée un codage de Prüfer noté `P` et sa taille notée `n` :

```
Créer un graphe G de n+2 sommets sans arêtes
Créer un tableau D de n+2 valeurs toutes initialisées à 1
Pour x parcourant P
  Augmenter D[x] d'un
Pour x parcourant P
  Trouver le plus petit indice d'un 1 dans D, noter j cet indice
  Relier x et j dans G
  Diminuer D[x] et D[j] d'un
Relier les deux derniers sommets dans G correspondant aux indices où il restait un 1 dans D
Renvoyer G
```

Deuxième algorithme, de mêmes entrées :

```
Créer un graphe G de n+2 sommets sans arêtes
Créer le tableau I des n+2 premiers entiers naturels
Pour i de 0 à n-1
  Soit x l'élément d'indice i de P
  Trouver le plus petit élément j de I qui n'apparaît pas entre l'indice i et l'indice n-1 de P
  Relier x et j dans G
  Retirer j de I
Relier les deux derniers sommets dans G correspondant aux éléments restants dans I
Renvoyer G
```

Question P1.7 : Implémenter le premier algorithme de décodage d'un codage de Prüfer.

Question P1.8 : Implémenter le deuxième algorithme de décodage d'un codage de Prüfer.

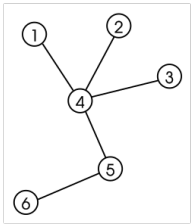
Question P1.9 : Démontrer que le décodage de l'une des deux questions précédentes donne bien un arbre.

Question P1.10 : En déduire la formule de Cayley : le nombre d'arbres-graphes non orientés à n sommets étiquetés par les premiers entiers naturels est n^{n-2}.

Attention : dans la formule de Cayley, la numérotation des sommets compte, il ne s'agit pas que de la forme. Deux arbres-graphes sont égaux à condition d'avoir la même matrice d'adjacence.

Question P1.11 : Déterminer la complexité en temps dans le pire des cas du codage de Prüfer avec les fonctions effectivement écrites dans la copie. De même pour chaque décodage effectivement écrit.

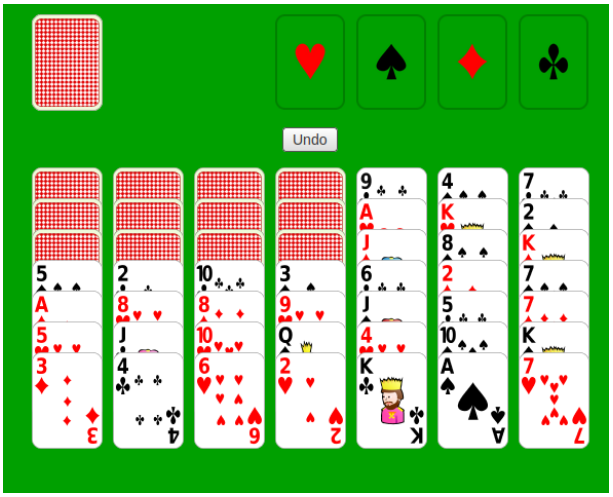
Question P1.12 : Mettre en œuvre le codage de Prüfer sur l'arbre représenté ci-après.



Question P1.13 : Mettre en œuvre le décodage de Prüfer (algorithme au choix) sur le codage 1 2 0.

Problème 2 : Scorpion solitaire [intégralement en OCaml].

Le jeu de Scorpion solitaire est un jeu de cartes de la famille des réussites / solitaires utilisant un paquet de cinquante-deux cartes disposé initialement au hasard comme dans la figure ci-après (source : `https://cardgames.io`).



Les règles de déplacement sont les suivantes :

- On peut déplacer une carte visible sur la carte **de la même couleur** (parmi pique, trèfle, cœur et carreau) et de valeur directement supérieure (ordre : as - deux - trois - . . . - dix - valet - dame - roi), à condition que cette dernière soit visible, ne soit couverte par aucune carte et soit sur une colonne différente. Ce faisant, toutes les cartes recouvrant la carte déplacée sont déplacées avec elle et continuent de la recouvrir dans le même ordre.
- On peut déplacer un roi sur une colonne vide, en déplaçant également les cartes recouvrant le roi selon le principe annoncé précédemment. **Pour simplifier le jeu et quitte à perdre une possibilité de gain, on interdit à un roi d'être déplacé depuis une colonne vide vers une autre.**
- Si une colonne a une carte invisible que plus rien ne recouvre, la carte recouvrant les autres est retournée. Dans l'illustration, ce serait le cas si on déplaçait le cinq de pique, le deux de trèfle, le dix de trèfle ou le trois de pique.
- Si aucun mouvement n'est possible et la pioche n'a pas encore été utilisée, les trois cartes de la pioche sont placées sur les trois premières colonnes de manière à les recouvrir.

Une fois qu'aucun mouvement n'est possible (sauf changer un roi de colonne alors qu'il est déjà au fond d'une colonne), la partie est gagnée si, et seulement si, les quatre couleurs sont organisées dans l'ordre chacune sur une colonne.

Dans l'optique d'une résolution du jeu, nous supposerons pour toute cette partie que les cartes invisibles sont connues (elles ne sont cependant pas disponibles).

Question P2.1 : Quels sont les mouvements possibles en ce début de partie ?

Question P2.2 : Combien peut-il y avoir au maximum de mouvements différents à un moment donné, à équivalence près ? Justifier.

Pour modéliser le jeu, une possibilité est de créer un tableau de tableaux de cartes (un tableau par colonne, l'indice maximal dans ce tableau correspondant à la carte qui n'est recouverte par aucune autre, et un tableau pour la pioche), en traitant de manière pertinente le fait qu'une carte soit visible ou non. Afin de cadrer le travail dans cette partie, on impose le type pour les cartes (type enregistrement à deux attributs numériques, de sorte que les cartes soient encodées par un entier de `0` à `12` pour la valeur et un entier de `0` à `3` pour la couleur) et la gestion de la visibilité par un tableau d'entiers déterminant le nombre de cartes cachées à la base de chaque colonne (malgré la configuration de départ, ce nombre sera renseigné pour les sept premières colonnes). On impose également que la pioche soit un tableau représenté en tant que variable supplémentaire absente du tableau.

```
type carte = { valeur : int ; couleur : int }

type jeu = { plateau : carte array array ; mutable pioche : carte array ; cachees : int array }
```

Question P2.3 : Écrire une fonction nommée `shuffle` prenant en argument un tableau et le mutant par un mélange, en respectant au maximum l'équiprobabilité de toutes les permutations possibles.

Pour la fonction précédente, on rappelle que la fonction `int` du module `Random` renvoie un entier pseudo-aléatoire entre zéro inclus et l'argument de la fonction exclu.

Question P2.4 : Écrire une fonction créant sans argument une configuration de départ du jeu et la renvoyant, sous la forme d'une variable de type `jeu`.

Question P2.5 : Écrire une fonction prenant en argument une configuration du jeu et renvoyant la liste des triplets `(i, j, k)` tels que la carte à la position `j` de la colonne numéro `i` du plateau puisse être déplacée sur la carte au sommet de la colonne numéro `k`. Cette fonction doit gérer le cas où la carte à déplacer est un roi et la colonne numéro `k` est vide, en excluant le cas où le roi est déjà en fond de colonne.

Question P2.6 : Écrire une fonction prenant en argument une configuration de jeu et déclenchant l'utilisation de la pioche. On ne vérifiera pas que les conditions d'utilisation de la pioche sont remplies.

Question P2.7 : Écrire une fonction prenant en argument une configuration de jeu et en renvoyant une copie.

Question P2.8 : Écrire une fonction prenant en argument une configuration de jeu et déterminant s'il existe une solution à ce jeu. On suggère de travailler par exploration exhaustive.

Question P2.9 : Prouver la terminaison de la fonction précédente et justifier qu'il n'est pas nécessaire de mémoriser les configurations rencontrées pour éviter les boucles ou récursions infinies (cette mémorisation serait excessive vu la mémoire nécessaire).

En pratique, travailler par exploration exhaustive peut s'avérer lourd dans le cas où il n'y a pas de solution et que cela peut se détecter rapidement. En effet, imaginons que le roi de trèfle soit directement recouvert par la dame de pique et que le roi de pique soit directement recouvert par la dame de trèfle. Alors pour déplacer la dame de trèfle il faut que la dame de pique soit déplacée au préalable et vice-versa. Ceci exclut en particulier qu'il y ait une solution.

Question P2.10 : Écrire une fonction prenant en argument **un plateau** de jeu et déterminant s'il n'y a pas de solution en raison de la présence d'un couple de cartes se bloquant mutuellement comme dans l'exemple ci-avant. On remarquera que le fait qu'une carte soit cachée ou non n'a pas d'influence sur le blocage, d'où le fait que l'argument soit limité au plateau.

Malheureusement, les blocages peuvent être plus compliqués à détecter, si par exemple ils proviennent d'un enchaînement de dépendances.

Nous allons utiliser une structure de graphe de dépendances associé à un plateau de jeu : les sommets seront les cartes (on peut se limiter aux cartes n'étant pas au fond d'une colonne ni absentes ni « bien placées ») et les arcs relieront la carte `C1` à la carte `C2` si la carte sur laquelle `C2` doit être déplacée est actuellement recouverte directement par `C1` (de sorte qu'il faille déplacer `C1` avant de pouvoir déplacer `C2`).

Question P2.11 : Écrire une fonction prenant en argument un plateau de jeu et construisant le graphe de dépendances associé. La représentation du graphe est au choix.

Question P2.12 : Écrire une fonction prenant en argument un plateau de jeu et déterminant, à l'aide du graphe de dépendances, s'il est impossible de résoudre le jeu. Si le graphe de dépendances ne permet pas de conclure à l'impossibilité de résoudre le jeu, on pourra répondre `false` même si en pratique le jeu est impossible à résoudre pour d'autres raisons.

Annexe

On rappelle à toute fin utile des fonctions du module `Hashtbl` avec des informations sur leur spécification :

- `Hashtbl.create n` crée une table de hachage avec `n` places pour commencer, mais en adaptant si besoin (donc on devine `n` sans qu'il n'y ait de risque si l'estimation est mauvaise) ;
- `Hashtbl.add th cle valeur` ajoute une association à la table de hachage, en masquant une éventuelle clé déjà existante (l'autre valeur sera de nouveau accessible en cas de retrait de ce qui l'a masqué) ;
- `Hashtbl.find th cle` détermine la valeur associée à la clé dans la table de hachage, en déclenchant l'erreur `Not_found` si la clé est absente ;
- `Hashtbl.mem th cle` détermine si la clé est présente dans la table de hachage ;
- `Hashtbl.remove th cle` retire une occurrence de la clé dans la table de hachage s'il y en a une (sinon la fonction n'a pas d'effet) ;
- `Hashtbl.replace th cle valeur` remplace la valeur associée à la clé dans la table de hachage par une nouvelle valeur (une éventuelle valeur masquée n'est pas impactée) en ajoutant la clé si elle n'y était pas encore.
- `Hashtbl.find_opt th cle` agit comme la fonction `find`, mais retourne une option pour éviter de lever une exception si la clé est absente ;
- `Hashtbl.iter f th` appelle la fonction fournie, prenant des clés et des valeurs (dans cet ordre) en argument, à tous les éléments de la table de hachage, sans contrôle sur l'ordre, sachant que si des clés sont masquées par d'autres clés identiques, elles subiront aussi la fonction (et on sait que ce sera dans l'ordre inverse de leur apparition dans la table de hachage).
- `Hashtbl.fold f th acc` appelle la fonction fournie, prenant des clés et des valeurs ainsi qu'un accumulateur (dans cet ordre) en argument, à tous les éléments de la table de hachage, sans contrôle sur l'ordre, sachant que si des clés sont masquées par d'autres clés identiques, elles subiront aussi la fonction (et on sait que ce sera dans l'ordre inverse de leur apparition dans la table de hachage). La fonction renverra la valeur de l'accumulateur à la suite de tous ces appels successifs.

On rappelle également des fonctions utiles du module `List` (noms et spécifications analogues pour les tableaux...) :

- `rev` renvoie la version retournée de la liste ;
- `for_all` et `exists` vérifient si tous les éléments (resp. au moins un élément) de la liste en deuxième argument satisfont (resp. satisfait) un prédicat, c'est-à-dire une fonction prenant en argument des objets du type commun des éléments de la liste, en premier argument, et retournant un booléen. La valeur `x` satisfait le prédicat `p` si, et seulement si, `p x` est vrai ;
- `mem` vérifie si le premier argument est un élément de la liste en deuxième argument.
- `map : ('a -> 'b) -> 'a list -> 'b list` applique son premier argument aux éléments de son deuxième argument dans l'ordre pour former une liste ;
- `iter : ('a -> unit) -> 'a list -> unit` fait la même chose, mais le premier argument est une fonction renvoyant un `unit` ;
- `filter : ('a -> bool) -> 'a list -> 'a list` renvoie la liste, dans le même ordre d'apparition, des éléments de la liste en deuxième argument qui satisfont le prédicat en premier argument.
- `fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b` applique son premier argument de manière imbriquée à tous les éléments de son deuxième argument conjointement à son troisième argument [1] ;
- `fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` est similaire, mais l'ordre est inversé [2].

1. En clair, `fold_right f [a1; ...; an] b` correspond à `f a1 (f a2 (...(f an b) ...))`.
2. De même, `fold_left f b [a1; ...; an]` correspond à `f (...(f (f b a1) a2) ...) an`.