TP14: Stratégies algorithmiques

1 Échauffement: gloutons (à faire en OCaml)

Exercice 1: Monnaie

Implémenter l'algorithme glouton vu en cours pour le rendu de monnaie:

```
(* rendu_glouton m a renvoie un tableau x de même taille que a,
   tel que chaque x.(i) donne le nombre de pièces a.(i) utilisées
   pour rendre la monnaie m gloutonnement.
   Précondition: les pièces sont triées par ordre croissant. *)
rendu_glouton: int -> 'a array -> 'a array
```

Exercice 2: Intervalles disjoints

On rappelle le problème des intervalles disjoints: étant donnés n intervalles $E_1, \dots E_n$, trouver le plus grand nombre d'évènements deux à deux disjoints.
L'algorithme glouton optimal vu en cours consiste à trier les intervalles par date de fin croissante, et à considérer les évènements un par un.

Q1. On note $E_i = (d_i, f_i)$ pour $i \in [\![1, n]\!]$. Soient $i_1 < \dots < i_k \in [\![1, n]\!]$ tels que $E_{i_1}, \dots, E_{i_k}$ sont deux à deux disjoints. Pour $i \in [\![i_k + 1, n]\!]$, proposer un critère simple permettant de déterminer si E_i est compatible avec tous les évènements $E_1, \dots, E_k$ (simple voulant ici dire calculable en $\mathcal{O}(1)$).

On représentera les intervalles par le type suivant:

```
type intervalle = int * int (* (début, fin) *)
```

La fonction `Array.sort` du module `Array` permet de trier un tableau en place:

```
Array.sort : ('a -> 'a -> int) -> 'a array -> unit
```

Le premier argument est la fonction de comparaison à utiliser: elle doit renvoyer un entier négatif, nul ou positif selon l'ordre des deux éléments comparés. Cet argument permet de choisir l'ordre à considérer pour le tri, ce qui évite de devoir forcément utiliser l'ordre par défaut d'OCaml.

Q2. Écrire une fonction de comparaison pour les intervalles permettant de les classer par ordre de fin croissante:

```
(* compare_intervalle e1 e2 renvoie:
   - -1 si la fin de e1 est avant celle de e2
   -  1 si la fin de e1 est après celle de e2
   -  0 si les évènements e1 et e2 finissent en même temps *)
compare_intervalle : intervalle -> intervalle -> int
```

Q3. Vérifier que pour `e` un tableau d'évènements, l'instruction suivante trie bien `e` par ordre de fins croissantes:

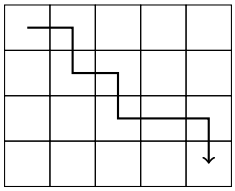
```
Array.sort compare_intervalle e
```

Q4. Implémenter l'algorithme glouton pour le problème des évènements disjoints:

```
(* glouton_evenements e détermine un ensemble de taille maximale d'évènements
   de e deux àdeux disjoints.
   Renvoie un tableau de booléens c de même taille que e, tel que
   c.(i) = true si l'on choisit e.(i) dans la solution, false sinon. *)
glouton_evenements : intervalle array -> bool array
```

2 Nettoyage de grille (à faire en C)

Le jardin Rectangulo est organisé selon une grille de taille $n \times m$. Chaque matin, pour aller au travail, vous entrez dans le parc Rectangulo par la case en haut à gauche de cette grille, de position $(0, 0)$, et vous sortez par la case en bas à droite, de position $(n-1, m-1)$. Pour cela, vous pouvez vous déplacer d'une case à la fois, soit vers la droite, soit vers le bas. Vous ne pouvez ni vous déplacer en diagonale, ni faire de détour vous ralentissant. Par exemple:



Suite à une coupe budgétaire, la ville ne peut plus payer le nettoyage régulier du parc. Sur chaque case de la grille se trouve un certain nombre de déchets. Ces déchets sont modélisés par une grille G telle que $G[i][j]$ est la quantité de déchets sur la case (i, j) de la grille. Vous voulez ramasser **le plus de déchets possible** en allant au travail, pour garder le parc propre.

Parsing

On stocke les instances du problème dans des fichiers textes, contenant:

- Sur la première ligne, deux entiers n, m indiquant les dimensions de la grille
- Sur les n lignes suivantes, m entiers séparés par des espaces indiquant la quantité de déchets sur chaque case.

Dans l'archive du TP, téléchargeable sur Cahier de Prépa, vous trouverez deux fichiers `grille.h/grille.c` contenant un type simple pour les grilles ainsi que des fonctions de lecture depuis un fichier texte au format décrit ci-dessus, d'affichage, et de libération mémoire. L'archive contient également trois fichiers exemple:

- Une petite grille de taille 4×5;
- Une grille moyenne de taille 23×34, dont la valeur optimale est 1189 et dont une version se trouve en ligne sur perso.ens-lyon.fr/guillaume.rousseau/mp2i/ramasse_miette/ (Ne passez pas plus de 3 minutes à faire joujou SVP);
- Une grande grille de taille 400×750 dont la valeur optimale est 25918.

Q1. Déterminer à la main une solution optimale pour la petite grille.

Pour représenter un chemin, on propose d'utiliser des chaines de caractères. Un `'D'` (comme Down) signifiera que l'on va vers le bas, et un `'R'` (comme Right) que l'on va vers la droite. Vous pouvez rentrer ces chaînes sur la page internet donnée plus haut pour visualiser vos chemins et vérifier vos résultats.

Q2. Écrire une fonction `int valeur(grid_t* g, char* chemin)` calculant la valeur d'un chemin dans une grille. Si le chemin ne contient pas que des `'D'` et des `'R'`, s'il sort de la grille, ou s'il n'atteint pas la case en bas à droite, la fonction renverra -1 et affichera un message d'avertissement.

Q3. Écrire une fonction `char* chemin_aleatoire(grid_t* g)` qui génère un chemin aléatoire entre $(0, 0)$ et $(n-1, m-1)$. On ne demande pas à ce que la fonction choisisse chaque chemin avec une probabilité uniforme. On pourra utiliser deux variables stockant le nombre de pas restants à faire vers le bas et vers la droite.

Q4. Lancez la fonction précédente plusieurs fois sur chacune des trois grilles. Vérifiez que vous obtenez des chemins valides, et aléatoires. Vérifiez que les solutions trouvées changent à chaque exécution, et notez les quelques valeurs, en vue de comparer avec des algorithmes plus efficaces.

Q5. Un peu de dénombrement: quelle serait la complexité de l'algorithme naïf consistant à énumérer tous les chemins ?

Glouton

Un algorithme glouton naturel pour ce problème est de construire un chemin petit à petit, en partant de la case de départ et en allant à chaque fois sur la case contenant le plus d'ordures parmi les deux cases adjacentes.

Q6. Écrire une fonction `bool choix_glouton(grid_t* g, int i, int j)` qui renvoie true si à partir de la case (i, j), l'algorithme glouton choisit d'aller sur la case à droite, et false s'il choisit la case en bas. Cette fonction prendra en compte les cas où la case (i, j) se trouve sur un des bords de la grille.

Q7. Écrire une fonction `char* chemin_glouton(grid_t* g)` qui renvoie une chemin généré par l'algorithme glouton.

Q8. Vérifiez que l'algorithme renvoie des chemins valides. Notez les résultats trouvés pour les trois grilles. Sont-ils meilleurs que les résultats aléatoires ?

Q9. Trouvez à la main un exemple simple où l'algorithme n'est pas optimal. (*Indication: essayez de piéger l'algorithme en l'emmenant dans une impasse !*)

Programmation Dynamique

Pour $0 \leq i < n$ et $0 \leq j < m$, on note $C(i,j)$ le nombre maximal de déchets que l'on peut ramasser en allant de $(0,0)$ à (i,j) (inclus)

Q10. Combien vaut $C(0,0)$?

Q11. Exprimez $C(i,j)$ en fonction de $C(i-1,j), C(i,j-1)$ et $G[i][j]$.

On stocke les valeurs de $C(i,j)$ dans un tableau à deux dimensions. On s'intéresse à une approche de bas en haut, où l'on remplit itérativement le tableau en question à l'aide de boucles.

Q12. Dans quel ordre peut-on remplir le tableau ?

Q13. Écrire une fonction `int** dechets_progdyn(grid_t* g, int n, int m)` qui calcule et renvoie le tableau de programmation dynamique décrit ci-dessus.

Q14. Vérifier que votre fonction trouve bien les valeurs optimales pour les trois grilles données dans l'archive.

Reconstruction du chemin

Pour reconstruire le chemin à partir du tableau de programmation dynamique calculé, on commence par se placer en $(n-1, m-1)$. On remonte alors jusqu'à la case $(0,0)$ en déterminant à chaque fois si l'on vient du haut ou de la gauche.
Notons que l'on a pas besoin de stocker dans une structure à part les indications permettant de reconstruire le chemin, car on peut refaire les calculs directement sur le tableau de programmation dynamique en $\mathcal{O}(1)$ par étape.

Q15. Écrire une fonction `char* reconstruction(int** D, grid_t* g)` qui reconstruit un chemin optimal dans g en utilisant le tableau D, qui contiendra le résultat de la programmation dynamique. Cette fonction affichera la quantité totale de déchets ramassés.

Q16. Vérifiez sur l'exemple en ligne que vous trouvez bien un chemin correct et optimal (sa valeur doit être 1189).

Bonus: variantes

Étudions quelques variantes du problème, en tentant à chaque fois de voir comment évolue la formule de récurrence. Chaque variante repart du problème initial, on ne les accumule pas.

Contrainte de place

Il y a plus de déchets que prévu, et vous vous rendez compte que votre sac poubelle ne peut pas contenir une infinité de déchets: il a une contenance K maximale. Cependant, vous détestez les déchets, et vous vous vous sentez obligés de ramasser tous les déchets des cases que vous visitez. Si votre sac ne peut pas contenir les déchets d'une case que vous visitez, vous êtes trop triste et vous rentrez chez vous. Vous devez donc choisir un chemin qui maximise la quantité de déchets ramassés **sans en ramasser strictement plus que** K.

On propose d'adapter la formule de récurrence obtenue à la partie précédente, en étudiant plutôt $C(i, j, k)$ la quantité maximale de déchets que l'on peut ramasser entre $(0, 0)$ et (i, j) en ayant un sac de contenance $k \leq K$, avec comme convention que $C(i, j, k) = -\infty$ s'il n'est pas possible d'aller de $(0, 0)$ à (i, j) avec un sac de contenance k.

Q17. Donner les cas de base et les formules de récurrence sur $C(i, j, k)$.

Q18. Justifier que la programmation dynamique **de haut en bas** est plus adaptée ici, et implémenter un algorithme permettant de résoudre le problème avec cette nouvelle contrainte.

Q19. Vérifier sur l'exemple en ligne qu'avec un sac de contenance 1000, on peut trouver un chemin permettant de ramasser exactement 1000 déchets.

Penser avec des portails

Le progrès technologique vous permet d'utiliser des portails à usage et sens unique. Vous disposez au départ de P portails. Vous pouvez à n'importe quel moment utiliser un des portails pour vous téléporter sur n'importe quelle autre case, **du moment que ce saut ne ralentit pas votre trajet** (interdit de se téléporter en arrière).
Pour (i, j) une position de case et $0 \leq p \leq P$, on note $C(i, j, p)$ le nombre maximal d'ordures que l'on peut ramasser entre $(0, 0)$ et (i, j) en utilisant au plus p portails, sans en ramasser plus que k.

Q20. Étudier $C(i, j, p)$ en donnant ses cas de base ainsi qu'une formule de récurrence.

Q21. Quelle est la complexité d'un algorithme de programmation dynamique découlant de la formule précédente ?

Zigzag

Dans cette variante, il est interdit d'avancer de strictement plus que trois pas de suite dans une direction donnée. Par exemple, à partir de la case de départ $(0, 0)$, vous pouvez marcher vers $(0, 1)$, $(0, 2)$, $(0, 3)$, mais **pas** $(0, 4)$: une fois arrivé en $(0, 3)$, il faut tourner et aller en $(1, 3)$.

Q22. Proposer une quantité à étudier, et trouver ses cas de base ainsi qu'une formule récursive. Quelle est la complexité de l'algorithme de programmation dynamique associé ?

Tout en même temps

Q23. Combiner les trois variantes.