TP5 : Programmation dynamique

Le but de ce TP est d'implanter des algorithmes de programmation dynamique vus en cours. Toutes les compilations nécessaires seront effectuées via la commande `make`.

Exercice 1. *Distance*

Le but de cet exercice est d'implanter le calcul de la distance d'édition vu en cours, ainsi que l'alignement de deux mots. Sont fournis pour cet exercice le fichier `edition.cc` qui contient le programme principal (à modifier) ainsi que le fichier `matrice.h` qui contient des fonctions pour déclarer, initialiser et afficher une matrice (à ne pas modifier). Le programme principal s'utilise comme suit : `./edition <mot1> <mot2>` calcule la distance entre `<mot1>` et `<mot2>`.

1. Compléter la fonction `int min(int a, int b, int c)` qui renvoie le minimum de ses trois arguments.
2. Compléter la fonction `int valeur(int** E, int i, int j)` : cette fonction doit renvoyer `E[i][j]` si $i \geq 0$ et $j \geq 0$, `i+1` si $j = -1$ et `j+1` si $i = -1$.
3. Compléter la fonction `int distanceEdition(string s1, string s2, int** E)` pour remplir la matrice E. *On utilisera la fonction* `valeur` *pour éviter d'avoir à tester si* $i-1$ *ou* $j-1$ *est négatif.*
   Tester la fonction en exécutant `./edition <mot1> <mot2>` avec différents couples de mots. Par exemple, `./edition AGORRYTTNES ALGORITHMES` doit afficher :

```
5
AGORRYTTNES

ALGORITHMES
```

4. Compléter la fonction `string alignement(string& s1, string& s2, int** E)` qui calcule un alignement des deux chaînes `s1` et `s2` : la fonction doit modifier `s1` et `s2` en insérant le caractère `_` (*underscore*) dans `s2` pour marquer une suppression, ou dans `s1` pour marquer une insertion. **Ne pas tenir compte dans cette question de la chaîne** `aligne` **qui est renvoyée.**
   Note. La fonction `s.insert(i, ch)` *insère la chaîne* `ch` *en position* `i` *de la chaîne* `s` *: par exemple si* `s = "mer"` *et qu'on applique* `s.insert(1, "ang")`*,* `s` *devient* `"manger"`.
   Tester votre fonction : `./edition AGORRYTTNES ALGORITHMES` doit maintenant afficher :

```
5
A_GORRYTTNES

ALGO_RITHMES
```

5. Finir de compléter la fonction `alignement` pour construire la chaîne `aligne` : celle-ci doit être constituée des quatre caractères `R` (remplacement), `I` (insertion), `S` (suppression) et `|` (barre verticale, pour signifier que les deux lettres alignées sont égales), afin d'indiquer pour chaque caractère des deux chaînes alignées du type de transformation (ou absence de transformation) effectuée.
   Par exemple, `./edition AGORRYTTNES ALGORITHMES` doit maintenant afficher :

```
5
A_GORRYTTNES
|I||S|R|RR||
ALGO_RITHMES
```

Exercice 2. *Voyageur de commerce*

Le but de cet exercice est d'implanter l'algorithme du voyageur de commerce vu en cours. Sont fournis pour cet exercice le fichier `tsp.cc` qui contient le programme principal (à modifier) et les fichiers suivants à ne pas modifier : `matrice.h` (comme dans l'exercice précédent), `ensembles.cc/.h` qui fournissent une implantation des sous-ensembles de $\{0,\dots,n-1\}$ par des entiers (*cf.* explications dans `ensembles.h`) et `points.cc/.h` qui fournissent des opérations pour générer des points dans le plan aléatoirement et dessiner le résultat dans des fichiers au format Postscript. Un point du plan sera représenté par un `coord` (qui est un `struct`, *cf.* `points.h`).

Le programme principal s'utilise ainsi : `./tsp 0` applique l'algorithme sur un ensemble fixé (*cf.* Fig. 2) ; `./tsp <n>` génère aléatoirement `<n>` points du plan et résout le problème du voyageur de commerce sur ces points (si $n > 0$).

Par simplicité, on implémente une **variante de l'algorithme TSP [1] présentée en Fig. 1 (p. 3)**.

1. Compléter la fonction `float distance(coord* P, int i, int j)` qui calcule la distance entre les points `P[i]` et `P[j]`.
2. Compléter le corps de la boucle `while` de la fonction `float tsp(int n, coord* P, float** D, int** Prec, int& min)` *en ignorant dans un premier temps les paramètres* `Prec` *et* `min` : on utilise la valeur `FLT_MAX` pour représenter $+\infty$; la boucle `while (U) { ... ; U = suivant(U, n-1, s); }` permet de parcourir tous les sous-ensembles $U \subset \{0,\dots,n-2\}$ de taille s.
   Tester : à ce stade, `./tsp 0` doit afficher

```
5 points : (601,423), (474,600), (490,397), (529,34), (45,365)
Longueur du circuit le plus court : 3.40282e+38
Circuit le plus court : 0 → 1 → 2 → 3 → 4 → 0

0 0 0 0 0 0 4.62428e-44 0 559.017 0 0 0 3.40282e+38 0 4.62428e-44 0
559.017 0 0 0 3.40282e+38 0 4.62428e-44 0 0 489.148 0 0 3.40282e+38 0 4.62428e-44 0
0 489.148 0 0 3.40282e+38 0 4.62428e-44 0 706.997 776.866 0 0 3.40282e+38 0 4.62428e-44 0
706.997 776.866 0 0 3.40282e+38 0 4.62428e-44 0 0 0 446.149 0 3.40282e+38 0 4.62428e-44 0
0 0 446.149 0 3.40282e+38 0 4.62428e-44 0 560.153 0 673.021 0 3.40282e+38 0 4.62428e-44 0
```

---

3. Continuer à compléter la fonction `tsp` en calculant le minimum final *en ignorant toujours les paramètres* `Prec` *et* `min`. **Tester :** à ce stade, `./tsp 0` doit calculer comme longueur `1772.45`.
4. Finir de compléter la fonction `tsp` en remplissant le tableau `Prec` et en conservant l'indice du minimum final dans `min`.
5. Compléter la fonction `void tsp_rec(int* circuit, int n, float** D, int** Prec, int min)` qui remplit le tableau `circuit` par la solution optimale du problème. **Tester :** à ce stade, `./tsp 0` doit calculer le circuit `4 → 1 → 0 → 2 → 3 → 4` et le fichier `Circuit.ps` doit être conforme à la Fig. 2.
6. *(bonus)* Compléter la fonction `float tsp_bruteforce(int n, coord* P)` qui calcule une solution optimale en testant toutes les permutations possibles des points. *On pourra utiliser la fonction* `next_permutation` *de la bibliothèque* `algorithm` *dont on trouvera une documentation à l'adresse* `https://fr.cppreference.com/w/cpp/algorithm/next_permutation`.
   Tester en décommentant la partie idoine du programme principal.

1. C'est un bon exercice que de comprendre pourquoi cette variante fait exactement le même calcul !

Algorithm: TSP(S)

$\Delta \leftarrow$ tableau à deux dimensions, indexé par les sous-ensembles de;
    $\{0,\ldots,n-2\}$ et par les sommets de s_0 à s_{n-1};
$\Delta[\emptyset, s_{n-1}] = 0$;
pour $s = 1$ *à* $n-1$ **faire**
    pour *tous les* $U \subset \{0,\ldots,n-2\}$ *de taille* s **faire**
        $\Delta[U, s_{n-1}] = +\infty$;
        pour *tout* $s_j \in U$ **faire**
            $\Delta[U, s_j] = \min\{\Delta[U \setminus \{s_j\}, s_i] + \delta_{ij} : s_i \in U, i \neq j\}$;
        fin
    fin
fin
retourner $\min_j(\Delta[\{s_0,\ldots,s_{n-2}\}, s_j) + \delta_{j,n-1}$;

FIGURE 1 – Variante de l'algorithme TSP.

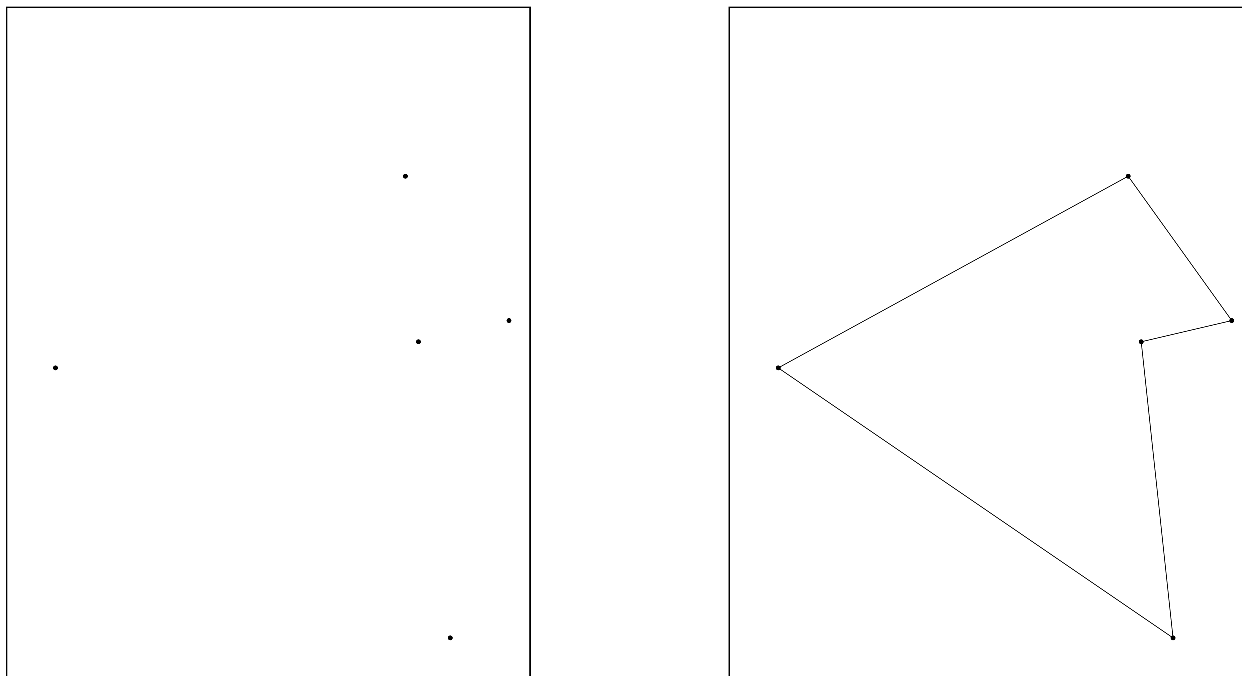

FIGURE 2 – Fichiers `Points.ps` (g) et `Circuit.ps` (d) pour l'exemple fixé de points.