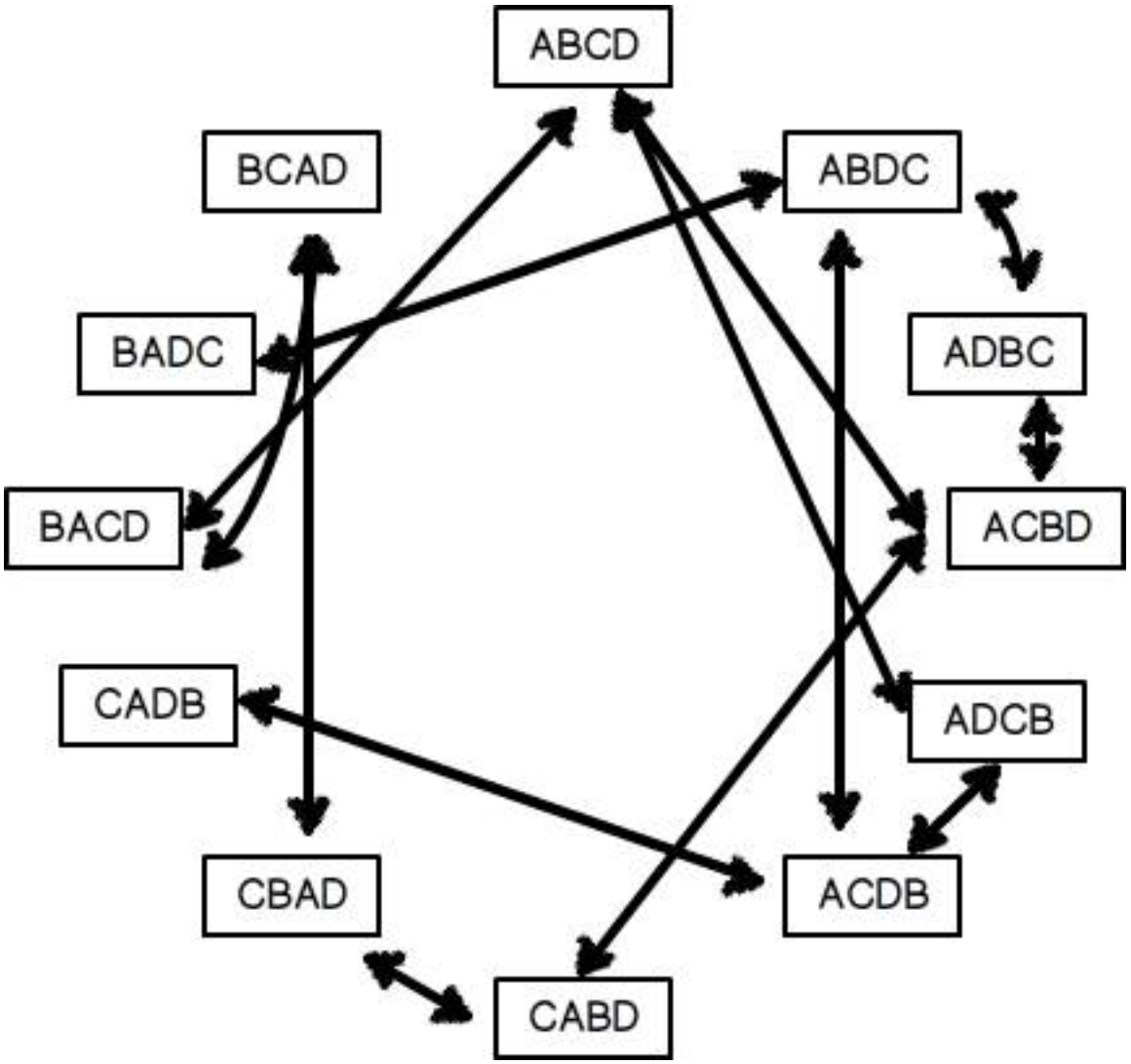
IA01 - TP n°2

Recherche dans un espace d'états

ROY Thibaut

HERMET Corentin

I – Graphe d'états



Les cases représentent les différents états possibles. Les flèches représentent les relations entre ces états, c'est à dire les échanges.

Il est vrai que la nature de l'échange n'a pas été détaillée mais il est facile de comprendre que la relation entre, par exemple, ADCB et ACDB est `(echange 2 3)`.

Ce graphe représente donc l'ensemble des états possibles atteignables à partir des échanges autorisés, c'est-à-dire des échanges 1-2, 2-3, et 2-4, et en respectant le fait que A doit être à gauche de D.

II – Fonction echange

```
(defun echange (p1 p2 etat)
      (let ((result (copy-list etat)) (len (length etat)))
            (if (and (<= 1 p1 len) (<= 1 p2 len))
                  (progn
                        (rotatef (nth (1- p1) result) (nth (1- p2) result))
                        result)))
```

Syntaxe d'utilisation : `(echange position1 position2 etat)`

Retour : nouvel état, résultant de l'échange. Si l'échange n'a pas été possible, retourne `nil`.

Explications :

Echange prend en paramètre la position du pion que l'on souhaite déplacer, sa position d'arrivée et l'état source. Elle renvoie le nouvel état après avoir fait l'échange.

L'opérateur <= vérifie que les éléments passés en paramètre soient croissants, cela nous permet de vérifier que la valeur passée en paramètre ne dépassent pas le nombre d'éléments dans l'état. Nous utilisons aussi la macro `rotatef` qui prend en paramètre un ensemble de paires 'place-place' et échange les valeurs de ces places. Cette macro ne retournant rien (`nil`) il est nécessaire de retourner le résultat de cet échange à la suite. Notons aussi l'utilisation de l'opérateur (1-) qui retourne une valeur inférieure de 1 par rapport à celle passée en paramètre, c'est un raccourci pour `(- p1 1)`. Enfin, notons aussi que puisque la macro `rotatef` agit directement sur le symbole, il est nécessaire de récupérer une copie d'état pour éviter d'éditer l'état d'entrée.

Attention, une première version de cette fonction testait la validité de cet échange par rapport à une liste d'actions autorisées. Il a été décidé que cette tâche serait effectué par la fonction successeurs à la place.

Tests :

```
* (echange 2 3 '(A D B C))
(A B D C)
* (echange 4 5 '(A D B C))
nil
```

Nous voyons bien que l'échange s'est produit s'il est possible. Sinon, il retourne nil.

III – Fonction successeurs

```
(defvar *actionsAutorisees* '((1 2) (2 3) (2 4)))

(defun successeurs (etat)
  (let (result)
    (dolist (a *actionsAutorisees* (reverse result))
      (let ((e (echange (car a) (cadr a) etat)))
        (if (and e (< (position 'A e :test #'equal)    (position 'D e :test #'equal)))
          (push e result))))))
```

Syntaxe d'utilisation : (successeurs '(A D B C)

Retour : Retourne la liste des successeurs possibles à partir de l'état donné en paramètre

Explications :

Cette fonction retourne la liste des successeurs possibles, c'est-à-dire tous les successeurs où A se trouve à gauche de D après avoir appliqué les différentes actions autorisées.

Dans notre cas, echange est appelé 3 fois et avec les paramètres position 1,2 / 2,3 / 2,4.

Pour cela, elle parcourt la liste prédéfinie des actions autorisées, effectue chacun d'entre elles et ajoute le résultat si le successeur est valide.

Notons l'utilisation du reverse afin d'inverser la liste, car push ajoute toujours en début de liste. Le premier élément réellement trouvé se trouve alors en fin de liste.

Tests :

```
* (SUCCESSEURS '(A B C D))
((B A C D) (A C B D) (A D C B))
* (SUCCESSEURS '(A D B C))
((A B D C) (A C B D))
* (SUCCESSEURS '(B A D C))
((A B D C))
```

Nous observons que les successeurs retournés sont bien tous valides car ils correspondent aux successeurs sur le graphe de la question 1.

IV / V – Algorithme et fonction de recherche en profondeur

Nous cherchons une fonction avec parcours en profondeur permettant d'afficher le premier chemin valide rencontré.

Algorithme :

Entrées : etat, but, chemin (optionnel)

Début recherche
Si chemin n'existe pas Alors
    Chemin <- ()
Fin si
Chemin <- chemin + etat
Si etat = but Alors
    Retourner le chemin inversé
Sinon
    Tant que S n'est pas vide et que OK est vide
        Si le premier membre de S fait parti du chemin
            Alors on l'enleve a S
            Sinon OK <- recherche(premier élément enlevé à S, but, chemin)
        Fin si
    Fin Tant que
Fin si
Retourne OK
Fin recherche

Fonction :

```
(defun recherche (etat but &optional chemin)
  (let ((c (if chemin chemin ())) OK (s (successeurs etat)))
    (push etat c)
    (if (equal etat but)
      (reverse chemin) ; si etat = but on retourne le chemin inversé
      (progn
        (loop while (and s (not OK)) ; tant que s non vide et OK vide
          do (if (member (car s) c :test #'equal) ; si premier élément de S ∈ chemin
            (pop s) ; on l'enleve de s et on loop
            (setf OK (recherche (pop s) but c)))) ; sinon on set OK par récursivité
        OK))) ; et on retourne OK
```

Syntaxe d'utilisation : (recherche etatInitial etatRecherché)

Retour : « Fin »

Explications :

Recherche prend en paramètre l'état courant, l'état recherché et le chemin des états parcourus (nul au début). Elle affiche le premier chemin possible rencontré

Un paramètre est optionnel afin d'éviter à l'utilisateur l'ajout en paramètre d'une liste vide inutile. Ce paramètre est toutefois utilisé pour nos appels récursifs afin de garder en mémoire les états déjà analysés.

Durant l'appel initial, le chemin est initialisé grâce à la ligne suivante : `((c (if chemin chemin ()))`

Enfin, cette fonction parcourt donc tous les chemins possibles, puis s'arrête au premier chemin valide rencontré.

Notons aussi l'utilisation du reverse afin de retourner le chemin parcouru et ce dans le bon ordre.

Test :

```
* (RECHERCHE '(A D B C) '(C B A D))
((A D B C) (A B D C) (A C D B) (A D C B) (A B C D) (B A C D) (B C A D) (C B A D))
```

Recherche affiche donc le premier chemin menant à l'état recherché.

VI – Heuristique

Une fonction heuristique (ou une heuristique) est une fonction qui permet de classer les différentes alternatives dans un algorithme de recherche à chaque étape et permettant de choisir la branche à emprunter en se basant sur les informations disponibles.

Une heuristique idéale serait basée sur une connaissance des différents états possibles afin de trouver le chemin idéal parmi le graphe. Hors, nous partons du principe que cette question nous demande à réfléchir à une heuristique naïve, ne connaissant pas l'ensemble des états possibles mais uniquement les états successeurs possibles à l'état actuel.

Nous avons tout d'abord choisi de calculer le nombre de lettres mal placées par rapport à l'état final. Cela correspond au calcul de la distance de Hamming. Dans notre cas, cela correspond au nombre de lettres qui ne sont pas à leur place par rapport au but recherché.

Exemple : si nous cherchons ACBD et que nous avons ABCD, la distance est de 2 car C et B ne sont pas à leur place.

Le résultat nous paraissait concluant mais pouvait être encore amélioré.

Nous avons alors ajouté à cela la distance de Manhattan (écart entre la position actuelle de la lettre et sa position finale) pour chacune des lettres mal placées.

Bien entendu, cette heuristique n'est pas parfaite, car, comme nous le verrons plus tard, elle ne trouve pas le meilleur chemin possible pour tous les cas.

VII – Fonction choixEtat

Nous implémentons donc cette heuristique.

```
(defun choixEtat (listeEtats but chemin)
(let (result (distanceHamming 0) (distanceManhattan 0) (distanceActuelle 0) lettresMalPlacees)
    (dolist (e listeEtats)
      (setf
        ;; remise à zéro des variables
        distanceHamming 0
        distanceManhattan 0
        ;; calcul de la distance de hamming : nombre de lettres à la mauvaise place
        lettresMalPlacees
          (loop
            for i from 0 to (1- (length e))
            when (not (equal (nth i e) (nth i but)))
            do (incf distanceHamming) collect (nth i e)))
        ;; calcul de la distance de manhattan pour chaque lettre mal placées :
        ;; écart entre la position d'une lettre mal placée dans l'état actuel et
        ;; de la position de cette même lettre dans l'état recherché
        (dolist (l lettresMalPlacees)
          (setf distanceManhattan (+ distanceManhattan
            (abs (- (position l but :test #'equal)
              (position l e :test #'equal))))))
      ;; on compare les distances actuelles avec la précédente et on modifie la
      ;; valeur de retour en conséquence
      (if result
        (if (and (not (member e chemin :test #'equal))
          (< (+ distanceManhattan distanceHamming) distanceActuelle))
            (setf result e distanceActuelle (+ distanceManhattan distanceHamming)))
        (if (not (member e chemin :test #'equal))
          (setf result e distanceActuelle (+ distanceManhattan distanceHamming)))))
    result))
```

Syntaxe d'utilisation : (choixEtat listeEtatsSuccesseurs but)

Retour : etatChoisi

Explications :

Cette fonction prend en paramètre la liste des états à classer ainsi que le but à atteindre et retourne l'état de la liste possédant le meilleur classement suivant notre heuristique.

Notons ici que la fonction setf agit sur des paires place/nouvelle-valeur, incf permet d'incrémenter la valeur d'une variable (En effet (+ 1 var) retourne 1+var mais la valeur de var reste la même.

Test :

```
* (CHOIXETAT '((A B D C) (A C B D)) '(C B A D))
(A C B D)
```

Nouvelle fonction de recherche :

```
(defun recherche (etat but &optional chemin)
      (let ((c (if chemin chemin ())))
            (push etat c)
            (if (equal etat but)
                  (reverse c)
                  (recherche (choixEtat (successeurs etat) but) but c))))
```

Fonction recherchant tous les chemins possible :

```
(defun recherche-all (etat but &optional chemin)
       (let ((c (if chemin chemin ())))
              (push etat c)
              (if (equal etat but)
                     (print (reverse c))
                     (dolist (x (successeurs etat) 'fin)
                            (if (not (member x c :test #'equal))
                                   (recherche-all x but c))))))
```

Test :

```
* (RECHERCHE '(A D B C) '(C B A D))
((A D B C) (A C B D) (C A B D) (C B A D))
* (RECHERCHE-OLD '(A D B C) '(C B A D))
((A D B C) (A B D C) (A C D B) (A D C B) (A B C D) (B A C D) (B C A D) (C B A D))
* (RECHERCHE-ALL '(A D B C) '(C B A D))
((A D B C) (A B D C) (A C D B) (A D C B) (A B C D) (B A C D) (B C A D) (C B A D))
((A D B C) (A B D C) (A C D B) (A D C B) (A B C D) (A C B D) (C A B D) (C B A D))
((A D B C) (A C B D) (C A B D) (C B A D))
((A D B C) (A C B D) (A B C D) (B A C D) (B C A D) (C B A D))
FIN
```

Ici, l'algorithme fonctionne correctement car on s'aperçoit que le chemin choisit est bien le plus court.

```
* (RECHERCHE '(A B C D) '(B A D C))
((A B C D) (B A C D) (B C A D) (C B A D) (C A B D) (A C B D) (A D B C) (A B D C) (B A D C))
* (RECHERCHE-OLD '(A B C D) '(B A D C))
((A B C D) (B A C D) (B C A D) (C B A D) (C A B D) (A C B D) (A D B C) (A B D C) (B A D C))
* (RECHERCHE-ALL '(A B C D) '(B A D C))
((A B C D) (B A C D) (B C A D) (C B A D) (C A B D) (A C B D) (A D B C) (A B D C) (B A D C))
((A B C D) (A C B D) (A D B C) (A B D C) (B A D C))
((A B C D) (A D C B) (A C D B) (A B D C) (B A D C))
FIN
```

Pour ce test, l'algorithme est moins efficace. Nous verrons la raison de cette inefficacité dans la partie suivante

VIII – Conclusion : comparaison des deux fonctions de recherche

Comme vu précédemment, cette heuristique n'est pas parfaite mais elle permet d'estimer le chemin qui semble le plus prometteur de manière correcte. Cela signifie que le chemin trouvé est POTENTIELLEMENT le plus court et elle fait le meilleur choix selon les données qu'elle possède. On ne peut donc pas deviner à tous les coups le chemin le plus court, ou alors on doit mettre en place un algorithme tel que celui de Dijkstra par exemple.

L'ancienne fonction de recherche nous affichait tous les chemins possibles. Nous pourrions modifier cette fonction afin qu'elle retourne le premier chemin rencontré, plutôt que de tous les afficher.

Dans ce cas, elle retournerait un long chemin pour ADBC -> CBAD alors que l'heuristique en renverrait un cours. Pour ABCD -> BADC les deux fonctions renverrait le même chemin. Cela s'explique car BACD (score = 4) est le successeur qui PARAIT le plus proche de BADC, comparé à ACBD (score = 10) et ADCB (score = 8).

Ainsi, la fonction de recherche avec l'heuristique est souvent plus efficace que celle qui n'est pas implémentée avec.